

*Advanced Perl Programming*



# 高级 Perl 编程

O'REILLY®

中国电力出版社

*Sriram Srinivasan* 著

*Perlish* 译

---

# 高级 Perl 编程

*Sriram Srinivasan* 著

*Perlish* 译

O'REILLY®

*Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo*

中国电力出版社



## 图书在版编目 (CIP) 数据

高级 Perl 编程 / (美) 斯里尼瓦桑 (Srinivasan, S.) 编著; Perlsh 译. - 北京: 中国电力出版社, 2001. 2

书名原文: Advanced Perl Programming

ISBN 7-5083-0512-4

I . 高 ... II . ①斯 ... ②P... III . Perl 语言 - 程序设计 IV . TP312

中国版本图书馆 CIP 数据核字 (2001) 第 02373 号

北京市版权局著作权合同登记

图字: 01-2000-3956 号

© 1997 by O'Reilly & Associates, Inc.

Simplified Chinese Edition, jointly published by O'Reilly & Associates, Inc. and China Electric Power Press, 2001. Authorized translation of the English edition, 2000 O'Reilly & Associates, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由 O'Reilly & Associates, Inc. 出版 1997。

简体中文版由中国电力出版社出版 2001。英文原版的翻译得到 O'Reilly & Associates, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者 —— O'Reilly & Associates, Inc. 的许可。

版权所有, 未得书面许可, 本书的任何部分和全部不得以任何形式重制。

书 名 / 高级 Perl 编程

书 号 / ISBN 7-5083-0512-4

责任编辑 / 刘江

封面设计 / Ellie Volckhausen, Hanna Dyer, 张健

出版发行 / 中国电力出版社

地 址 / 北京三里河路 6 号 (邮政编码 100044)

经 销 / 全国新华书店

印 刷 / 北京市地矿印刷厂

开 本 / 787 毫米 × 1092 毫米 16 开本 31 印张 480 千字

版 次 / 2001 年 2 月第一版 2001 年 2 月第一次印刷

印 数 / 0001-5000 册

定 价 / 69.00 元 (册)

MS28P/07

鄧子規

PDG

## 译者序

对一种计算机语言优越性的衡量不是看有多少人在使用它,而是看人们是否喜欢它。Perl被程序员称为“瑞士军刀”,有着众多坚定的支持者和拥护者,是一件为专业人员打造的利器。许多复杂应用在Perl程序员看来可以轻松搞定,极高的编程效率是他们常常面带优越感的原因。Perl博大精深,拥有自己的文化。Perl有着融合其他优秀语言的长处。

本书在Perl社团中被昵称为“Panther Book(黑豹书)”,是众多Perl黑客必读书中的一本(其他几本还包括《Programming Perl》,《Perl Cookbook》等,也即将由中国电力出版社陆续出版)。它描述了Perl的体系结构,涵盖了许多其他有关Perl的图书所没有涉及到的极有价值的专题,在每个专题结尾还将Perl与Python、Tcl、Java和C/C++语言进行了颇有趣味的比较,可以使你对Perl本身的理解上升到一个崭新的高度。作者对Perl的精通令人钦佩。针对每一个专题,作者不是单纯的讲述如何使用该语言去做,而是精辟的阐述了这样做的原因和内部机理。你从这本书中获得的不仅仅是对Perl的透彻理解,还能学到许许多多编程思想,和深入挖掘计算技术的方法和指导。译者本人就获益匪浅。希望将这本有价值的图书奉献给国内的Perl程序员们,能促进国内Perl社团的发展,让更多的Perl黑客有机会深入理解Perl,为CPAN做出自己的贡献。

由于时间仓促,译文中难免会有不妥甚至错误的地方,希望读者批评指正。

—— Perlsh

2001年元旦

资源分享网  
PDG

---

# 目录

前言 .....	1
<b>第一章 数据引用与匿名存储 .....</b>	<b>17</b>
对已有变量的引用 .....	19
使用引用 .....	26
嵌套数据结构 .....	32
引用的查询 .....	34
符号引用 .....	35
内部工作细节 .....	36
其他语言中的引用 .....	41
相关资源 .....	42
<b>第二章 实现复杂的数据结构 .....</b>	<b>43</b>
用户定义数据结构 .....	44
例子：矩阵 .....	45
教授，学生与课程 .....	49
颁奖 .....	53



格式化打印工具 .....	56
相关资源 .....	60
<b>第三章 Typeglob 和符号表 .....</b>	<b>61</b>
Perl 变量, 符号表和作用域 .....	62
Typeglob .....	66
Typeglob 与引用 .....	71
文件句柄, 目录句柄及打印格式 .....	73
<b>第四章 子例程引用与闭包 .....</b>	<b>77</b>
子例程引用 .....	78
使用子例程引用 .....	80
闭包 .....	83
闭包的应用 .....	86
和其他语言的比较 .....	92
相关资源 .....	93
<b>第五章 Eval .....</b>	<b>94</b>
字符串形式: 表达式计算 .....	95
代码块形式: 例外处理 .....	97
注意你的引号 .....	100
应用 Eval 来进行表达式计算 .....	101
应用 Eval 来提高运行效率 .....	103
在超时中应用 Eval .....	110
其他语言中的 Eval .....	112
相关资源 .....	114
<b>第六章 模块 .....</b>	<b>115</b>
包的基本知识 .....	116
包与文件 .....	118

包的初始化与销毁 .....	120
私有性 .....	121
符号的导入 .....	123
包的嵌套 .....	126
自动加载 .....	127
存取符号表 .....	128
与其他语言的比较 .....	130
<b>第七章 面向对象编程 .....</b>	<b>133</b>
面向对象简介 .....	133
Perl 中的对象 .....	135
UNIVERSAL .....	151
习惯的更新 .....	153
与其他面向对象语言的对比 .....	157
相关资源 .....	159
<b>第八章 面向对象：下面的几步 .....</b>	<b>161</b>
高效的属性存储 .....	161
代理 .....	174
关于继承 .....	175
相关资源 .....	178
<b>第九章 绑定 .....</b>	<b>179</b>
标量变量的绑定 .....	180
数组的绑定 .....	183
散列表的绑定 .....	186
文件句柄的绑定 .....	188
例子：对变量的监控 .....	189
与其他语言的比较 .....	194

<b>第十章 持续性 .....</b>	<b>196</b>
有关持续性的问题 .....	197
流式数据 .....	199
面向记录的方案 .....	202
关系数据库 .....	205
相关资源 .....	212
 <b>第十一章 对象持续性的实现 .....</b>	 <b>214</b>
适配器介绍 .....	216
设计注意事项 .....	219
实现 .....	226
相关资源 .....	236
 <b>第十二章 使用套接字进行网络编程 .....</b>	 <b>238</b>
网络计算入门 .....	238
Socket API 和 IO::Socket .....	240
同时处理多个客户端 .....	243
现实世界中的服务器 .....	249
IO 对象和文件句柄 .....	250
预编译的客户端模块 .....	252
相关资源 .....	254
 <b>第十三章 网络计算：RPC 的实现 .....</b>	 <b>255</b>
Msg:消息传递工具包 .....	255
远程过程调用 (RPC) .....	270
相关资源 .....	276
 <b>第十四章 使用 Tk 进行用户界面编程 .....</b>	 <b>278</b>
对 GUI, Tk 和 Perl/Tk 的介绍 .....	279
开始使用 Perl/Tk .....	280



组件之旅 .....	283
布局管理 .....	303
定时器 .....	307
事件联编 .....	307
事件循环 .....	310
相关资源 .....	312
 <b>第十五章 GUI 实例: Tetris.....</b>	<b>313</b>
有关 Tetris 的介绍 .....	314
设计 .....	315
实现 .....	316
 <b>第十六章 GUI 实例: Man 页面查看器.....</b>	<b>324</b>
Man 与 perlman .....	325
实现 .....	326
相关资源 .....	334
 <b>第十七章 模板驱动的代码生成.....</b>	<b>335</b>
有关代码生成的问题 .....	335
Jeeves 的例子 .....	339
Jeeves 概述 .....	344
Jeeves 的实现 .....	346
规格语法分析器样例 .....	355
相关资源 .....	357
 <b>第十八章 扩展 Perl: 第一课 .....</b>	<b>359</b>
编写一个扩展: 概述 .....	360
例子: Perl 与分形计算 .....	364
SWIG 的功能 .....	368
XS 的功能 .....	371

自由度 .....	375
分形介绍 .....	376
相关资源 .....	380
<b>第十九章 Perl 的嵌入：简单的方式 .....</b>	<b>381</b>
为什么要嵌入? .....	381
解释器嵌入概述 .....	383
例子 .....	386
增加扩展 .....	390
相关资源 .....	391
<b>第二十章 Perl 的内部工作 .....</b>	<b>392</b>
阅读源代码 .....	393
体系结构 .....	394
Perl 的值类型 .....	402
堆栈与消息传递协议 .....	429
内涵丰富的扩展 .....	436
简单的嵌入式 API .....	448
未来展望 .....	451
相关资源 .....	453
<b>附录一 Tk 组件参考 .....</b>	<b>455</b>
<b>附录二 语法概要 .....</b>	<b>472</b>
<b>词汇表 .....</b>	<b>481</b>

---

# 前言

错误,就像稻草,漂浮在水面上;  
搜寻珍珠的人必须潜入水下。

—— 约翰·德莱登 (译注1)

《一切为了爱·序曲》

写这本书有两个目的:一是使你成为 Perl 编程专家,一是在更广泛的意义上,为你补充编写应用系统的技能与工具。本书讲述了 Perl 语言的高级特性,教你学习 perl 解释器的工作原理,以及诸如网络计算、用户界面、对象的持续存储 (persistence) 与代码生成等现代计算技术。

从这本书中,你不仅仅会涉及 Perl 语言的语法和各种模块的 API (应用编程接口),你同样要花费相当的时间来处理现实情况中的问题,如在远程过程调用中避免死锁,在平面文件 (flat file) 或数据库的数据存储方式之间进行平滑的切换。同时,你将熟练使用诸如运行时 (run-time) 计算、嵌套数据结构、对象以及闭包 (closure) 一类的技巧。

在你读这本书以前,你需要了解 Perl 的基本知识,实际上只需要其中关键的一小部分就可以了。你必须熟知其基本数据类型 (标准变量、数组和散列表),正则表达式,子例程,基本控制结构 (if, while, unless, for, foreach), 文件的输入输出,以及一些标准变量如 @ARGV 和 \$\_ 等。如果你对 these 不甚了解的话,我向你推荐 Randal Schwartz 与 Tom Christiansen 所著的优秀辅导教程《Learning Perl》第二版 (译注2)。

---

译注1: 约翰·德莱登 (1631 - 1700), 英国著名诗人。

译注2: 中文版《Perl 语言入门》已由中国电力出版社出版,请访问 [www.infopower.com.cn](http://www.infopower.com.cn)。



这本书，特别是这段序言，将详细阐述我的两条理念。

第一条理念是，对于处理那些典型的大型应用系统工程而言，同时采用两种语言的方案，也就是将 Perl, Visual Basic, Python 或 Tcl 这样的脚本语言，同系统编程语言 (C, C++, Java) 结合起来的方案最为合适。脚本语言没有严格的编译时 (compile-time) 类型检查，拥有高级数据结构 (例如 Perl 中作为基本数据类型的散列表，C 语言中就没有这种东西)，一般没有附加的编译与链接过程。系统编程语言则往往更加贴近操作系统，具有精细的数据类型 (如 C 语言中就有 short, int, long, unsigned int, float, double 等等，而 Perl 中却只有标量变量类型)，通常要比解释语言运行速度快。Perl 拥有多种编程语言的许多特性。作为脚本语言它表现出色，然而它也提供低级的访问操作系统 API 的功能，其速度要比 Java 快许多 (这是指本书英文版出版时的情况)，而且需要的话还可以进行编译。

讨论脚本语言与系统编程语言的差异是一个有争议的话题。但是实践中我却因此获益匪浅。这一点将在最后三章中着重阐述。

我相信这两类语言没有一种具备足够的特性，可以独立处理复杂应用项目的开发。而且我希望能够将前面提到的结合两类语言的方案以 Perl 和 C/C++ 为例阐述清楚。即便是你选择使用其他语言，如果本书中所讲述的设计思想和教训对你有所帮助的话，那么就像孩子们常说的，“这样真值，这太棒了”。

我的第二条理念就是：要想部署更为有效的应用系统，仅熟知编程语言的语法是不够的。你必须更进一步了解语言的内部机制，必须扎实的掌握诸如网络、用户界面、数据库等技术领域的知识 (尤其是那些超出特定语言功能库的知识)。

让我们来更详细的阐述一下这两条理念。

## 脚本编程语言

我的职业生涯始于用汇编语言来创建整个应用系统。那时我不时担心的是如何节省 100 个字节的空间和优化删除一条指令。后来 C 和 PL/M 改变了我的世界观，使自己有机会从工程项目的生命周期 (life-cycle) 和最终用户如何使用等方面，从整体上把握应用系统。对于中断服务例程而言，当运行效率是最首要的需求时，

我依旧使用汇编语言（回首过去，我曾怀疑过 PL/M 编译器产生的汇编指令是否能比我手工编制的更为出色，我的虚荣心使我始终无法承认这一点）。

我参与的应用系统需求越来越复杂；除了要处理图形用户界面、事务、安全、网络无关和异构平台的问题，我又开始着手进行诸如航空调度和网络管理一类问题的软件体系结构设计。我自己的工作效率却要比应用系统的更成问题。虽然面向对象技术使我在设计上更有效率，而系统实现语言 C++，还有那些功能库和工具并没有帮助我提高编程水平。我仍旧需要处理诸如为动态数组、元数据、文本操纵和内存管理等创建应用框架之类的底层问题。不幸的是，能够很好的处理此类问题的计算环境如 Eiffel，Smalltalk 和 NeXT 系统对我的组织而言并不实用。你也许现在可以理解，为什么我会成为支持选择 Java 语言作为应用开发语言的嗓门嘶哑的啦啦队长了。尽管这还不是最终的结果。

近来我渐渐发现自己忽略了软件生命周期中的两大时间黑洞。在设计阶段，有时你要想清楚的理解问题，就需要创建“电子情节串联图版”（也就是原型）。在该软件完成后，用户通常很会对他们所看到的一切挑三拣四。这就意味着即便是基于窗体的简单界面也会被不时的修改，不停的产生新的需求报告。于是，那些急于求成的开发人员就希望在该软件的开发一完成就进入另一个项目。这里就是脚本语言的用武之地。它能提供快速的代码修改，动态用户界面，绝佳的文本处理功能，运行时计算和良好的数据库与网络连通性。最重要的是，它们不需要细致的程序员的精心呵护。你可以将注意力放在如何使应用更以用户为中心，而不是如何用 Xlib 库（注 1）来画饼图上。

显而易见，单独使用脚本语言来开发复杂的应用并不可行。你仍需要那些诸如运行效率、精细的数据结构和类型安全之类的特性。（这一点在多个程序员处理同一个问题时尤为重要。）这也是我热心支持将脚本语言同 C/C++（当性能达到实际应用水平时，Java 也是一种选择）结合起来使用的原因。许多人从这种基于组件（component-based）形式的开发中获益匪浅，这是一种用 C 语言来书写组件代码，然后使用脚本语言将组件连接起来的方法。你去问一下那些众多的 Visual Basic，PowerBuilder，Delphi，Tcl 和 Perl 程序员们就知道了。噢，对了，还有微软的 Office 和 Emacs 用户们。

注 1： X Window 函数库。有人曾经提到过，X Window 编程，就好像使用罗马数字算出一个数的平方根！

要寻找对使用脚本编程更为详尽而雄辩的、根据切身体会描述的文章（这里不提它们的争议性），你可以读一读 John Ousterhout 博士（注 2）的文章，地址在 <http://www.scriptics.com/people/john.ousterhout>。

想要更真切的体味一下这种论断的话，请试用一下在上面那个地址中提到的 Netscape 的 Tcl 插件，看一看 Tcl 小应用程序（Tclets）的代码，你会发现用它来解决一些简单问题是多么的简洁。一个包含用户界面的计算器小应用程序只花费了 100 行代码，恐怕写同样功能的 Java 小应用程序的代码量不会低于 800 行，而且远没有前者灵活。

## 为什么要选择 Perl 语言

那么为什么要选择使用 Perl 而不是 Visual Basic, Tcl 或 Python 呢？

尽管 Visual Basic 在 Wintel（注 3）的 PC 上是一种优秀的选择，但是它无法在其他平台上运行，因此对我来说，它就不是一种实际可行的选择。

Tcl 会使我更频繁的使用 C 语言，主要就是因为数据和代码结构的原因。Tcl 的性能对我来说并不是一种关键因素。因为我通常考虑到这一实际情况，所以只用它来书写那些对性能要求不高的部分。我向你推荐 Brian Kernighan（译注 3）的文章“Tcl/Tk 在科学与工程可视化中的应用经验谈”中有关 Tcl 和 Visual Basic 的论述。它的获取地址为 <http://inferno.bell-labs.com/cm/cs/who/bwk>。

大多数的 Tcl 用户基本上都离不开 Tk 用户界面工具包，我也不例外。在 Perl 中同样可以使用 Tk，因此我可以在自己首选的语言中使用另一种语言环境中的最为优秀的特性。

---

注 2: Tcl (Tool Command Language, 工具命令语言, 发音为“tickle”) 的发明者。

注 3: Wintel: 微软视窗操作系统 Windows 与 Intel 微处理器的组合。以后我将使用“PC”这个字眼来表示这种特定的组合，如果是指 Linux 和 Mac 机我会明确提出。

译注 3: 他是 C 语言的创造者之一。



我是个不折不扣的 Python 崇拜者。这是一种由 Guido van Rossum 开发的脚本语言（可以查看相关网址 <http://www.python.org/>）。Python 语言拥有清晰的语法结构，良好的面向对象机制和线程安全，拥有大量的功能库及与 C 语言的完美接口。我之所以更倾向于 Perl（相对 Python 而言）是出于实用而不是工程上的原因。在工程方面，Perl 语言在文本处理上快速而无可替代。其语言高度专业化，也就意味着比一般用其他语言编写出的代码更为紧凑。后者或许不是种好事，这要看你持何种观点（尤其对于 Python 程序员来说）。所有这些特性使 Perl 成为一种构造工具的优秀编程语言。（请参考第十七章“模板驱动的代码生成”。）在其他方面，Python 则拥有更多的优势。我建议你进行认真的分析。Mark Lutz 的《Programming Python》（O'Reilly 公司 1996 年出版）一书对 Python 语言和功能库进行了很好的描述。

从实用角度讲，你当地的书店和报纸上的招工信息已经表明了 Perl 语言的流行性。基本上说，这就意味着更容易雇佣到 Perl 程序员，或是人们可以很快的学习这种语言。我敢打赌，95% 的程序员甚至从没听说过 Python 语言（译注 4）。不幸的是，这种情况是真的。

把玩这些语言并得出你自己的结论是必要的；毕竟，前面的一些观察都带有我个人的色彩。正如 Byron Langenfeld 所说的：“很少有人能不经实地测量而发现别人的错误。”本书将把 Tcl、Python、Java 与 Perl 在具体特性上做一比较，我要强调的是，对语言和开发工具的选择，从来没有定论，也不是非此即彼的，我还要说明，大多数时候，你可以使用其中一种，也可以使用另一种。

## 我必须要了解什么？

要想在应用系统中更有效的使用 Perl 语言，你必须熟知以下三方面信息：

- 语言的语法及语言所提供的专业用语。
- Perl 解释器本身，这是为了书写 Perl 脚本程序的 C 扩展模块，或是将 Perl 解释器嵌入到你的 C/C++ 应用系统中。

---

译注 4：这是 1997 年本书首次出版时的情况，今天，Python 已经成了热门语言。

- 诸如网络、用户界面、万维网和持续性存储等相关技术问题。

图0-1描述了本书中要谈论的主题。每条所列的主要内容都进行了进一步细分。余下的章节描述了每个话题的简单介绍以及详细论述的章节。它们是按照主题而不是本书中章节出现的先后次序来组织的。

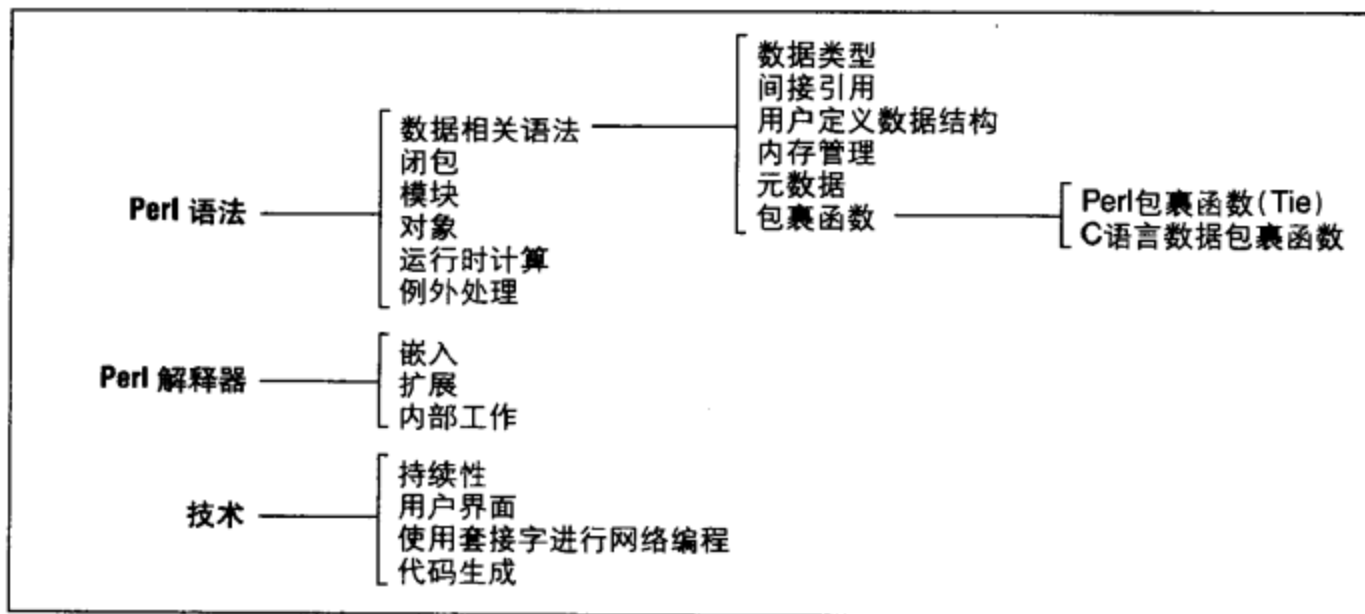


图 0-1 本书涵盖的主题分类

## 语言语法

指针（pointer）或引用（reference）可以使你创建非常复杂的数据结构。Perl 语言支持引用，而且还允许你无须进行烦琐的声明，即可直接书写代码，因此成为一种功能极其强大的编程语言。例如，你只用一行代码就能创建由数组的散列表所组成的数组（注4）。第一章“数据引用与匿名存储”，向你介绍了引用的概念以及 Perl 内部的内存管理机理。第二章“实现复杂的数据结构”通过几个实际的例子练习使用前面章节讲述的语法内容。

Perl 支持对子例程的引用和一种称为闭包的强大结构。LISP 程序员对此可能比较熟悉，这实际上是一种没有名字的子例程，通过上下文环境来交换信息。这种机

注4： 我们以后将把索引列表/数组（indexed list/array）称为“数组”，而将关联数组（associative array）称作“散列表（hash）”以避免混淆。

制及其相关专业用语将在第四章“子例程引用与闭包”中予以澄清并加以很好的利用。

引用只是一种间接存取的方式。标量变量中可以包含指向C数据结构的嵌入式指针。该内容放在第二十章“Perl的内部工作”中讲述。Tie代表着另外一种形式的间接存取：所有Perl的值在创建、存取或释放时都可以有选择的触发特定的Perl子例程。这些要在第九章“绑定”中讨论。

文件句柄、目录句柄和格式都不能算是基本的数据类型；它们之间不能相互赋值或作为参数传递，而且你不能够为它们创建局部的版本。在第三章“Typeglob和符号表”中我们将学习这些机制处于首要位置的原因，以及如何通过迂回的方式来获得这种机制。这一章中重点讨论一种数据类型typeglob，（它在某种程度上是隐含的）以及它的内部表示。理解这些内容对于获取解释器状态信息（元数据）以及创建方便的别名来说至关重要。

现在让我们来讨论与Perl数据类型并不直接相关的语言问题。

Perl支持包括异步例外（从信号处理过程中产生用户定义的例外的能力）在内的例外处理。实际上，eval除被用来进行运行时计算外，还应用在例外的捕获上。在第五章“Eval”中将对这两方面貌似不同而实际相关的话题进行讲解。

第六章“模块”中将详细讲述Perl对模块化编程的支持，其中包括的特性如运行时联编（也就是只有在运行时才决定调用哪个过程），继承（Perl透明的使用来自于另外一个类的子例程的能力），还有自动加载（捕获对不存在函数的调用并进行相应的动作）。第七章“面向对象编程”中将使模块的概念从逻辑上更进一层：不仅从功能库用户的观点体现模块的重用性，而且还从增加库接口的开发者的角度加以体现。

Perl支持运行时计算：也就是把字符串看作小段Perl程序，动态的计算它们的值。第五章中引入关键字eval，并举出一些例子来描述如何使用这种特性，但是它真正的重要性还要在后面的章节中详细讨论。它将被应用在诸如SQL查询的计算（第十一章“对象持续性的实现”），代码生成（第十七章“模板驱动的代码生成”）及动态生成对象属性的存取函数上（第八章“面向对象：下面的几步”）。

## Perl 解释器

有三个章节用以讲述使用和理解Perl解释器。如此深入研究Perl的内部工作情况，主要有两方面的原因。一个是为了扩展Perl，也就是通过书写C模块的方式，来完成不适合Perl来做的或者速度不够快的功能。另一个原因就是为了让C中嵌入Perl执行环境，这样，C程序就可以调用Perl，完成诸如处理正则表达式替换等，你或许不想通过编写C代码来完成的特定任务。

第十八章“扩展Perl：第一课”，讲解了两个用于扩展Perl解释器而创建动态加载C功能库的工具（xsubpp和SWIG）。

第十九章“Perl的嵌入：简单的方式”，讲解了一套专为本书开发的方便的应用编程接口（API），它可以使你无须考虑Perl的内部工作细节就能嵌入解释器。

但是，如果你真想了解那些内部的工作情况，或者想要开发功能强大的扩展的话，第二十章将满足你的渴求（根据你可能的需求来提供相应的细节）。

## 技术领域

我个人以为，作为一名应用系统开发人员，应当掌握至少以下六个主要技术领域的知识，它们包括用户界面，持续存储，进程间通信与网络，语法分析及代码生成，万维网技术和操作系统。本书对前四种技术进行了详细的讨论（从第十章到第十七章）。本书一开始就从实际问题出发并开发有用的解决方案，涵盖了相应的Perl软件包，而不只是向你讲解可用模块的API。例如，第十三章“网络计算：RPC的实现”，讲解了一种即使两个进程碰巧同时调用对方，也不会死锁的RPC工具箱的实现。另一个例子就是，第十一章“对象持续性的实现”开发了一种“适配器”，它将一组对象发送到你所选择的（如关系数据库、普通文件或DBM文件）的持续存储中，并实现了在任何一种方式上的查询操作。

本书并不讨论与操作系统相关的问题，部分原因是Perl隐藏了大量的操作系统方面的差异，部分原因则是因为这些细节将会干扰我们这本书的核心话题。实际上本书的所有代码都是独立于操作系统的。

我选择放弃那些与万维网有关，确切的说是与CGI有关的问题。这主要是因为市面上有大量有关使用Perl来编写CGI脚本的图书（注5）和指导教程，它们提供了比在本书有限的空间所能负担的更好的讲解内容。而且大多数有意义的CGI应用的开发人员，会将大量的时间花费在本书所讲解的概念而不是简单的CGI协议本身上。

## 本书的讲解方式

你购买这本书当然不仅仅是为了看一下Perl的一系列的特性。如果那样的话，免费的联机文档就足够了。我想通过本书传授的是，如何应用恰当的特性和前面所提到的基础技术领域的知识，来解决实际问题的技巧。

## 留给专家的话

本书采取了一种指导教程的方式来阐明Perl语法中的各类问题。先是提出某种概念或特性的必要性，然后再来解释Perl是如何对此提供支持的。对于那些不需我讨论什么特性或繁冗例子的经验丰富的人来说，或许更好的方式就是先浏览一下附录二“语法概要”，快速的了解一下本书所有的结构和专业用语，而后再在需要时去查看相应的讲解。

我热切希望本书中有关技术，嵌入，扩展和Perl内部工作细节的章节（与语法无关的章节）对普通用户和专家一级的人员同样有用。

## 系统观点

本书倾向与采取一种系统化的方式；大多数章节都有一段用来描述本章实质内容的文字。我相信如果你不了解编译和运行环境是如何实现的，而只知道些语言的语法规则的话，你不可能是一个好的程序员。比如，一位C程序员必须要知道从函数中返回局部变量是个坏习惯（而且也要知道这种限制的原因），还有就是一位

---

注5： 请参考 Shishir Gundavaram 的书《CGI Programming with Perl》（O'Reilly 公司出版），中文版即将由中国电力出版社出版。

Java程序员应当知道为什么一个线程即便没有被阻塞,也可能在单一处理器的环境中永远不能获得控制权。

此外,从根本上知晓这一切是如何工作的,将会使你对各种特性的理解更加持久。那些了解词汇词源的人,不用费多大力气就能维持极大的词汇量。

## 范例

Perl 是一种用语高度专业化的语言,具有丰富的冗余特性(注6)。虽然我像别人一样热情高涨的对待各种奇妙的探索语言的方式(注7),但本书并不是容纳各类特性的手册;它包含了用以开发强大应用的 Perl 所有功能的最小集合。

在所展示的范例代码中,我牺牲了精简和效率来换取代码的可读性。

## FTP

如果你能连接因特网的话(永久性连接或拨号式的),最简单的方法就是用浏览器或是自己喜爱的FTP客户端程序以FTP的方式获取范例程序代码。你只须简单的在浏览器中输入以下地址:

```
ftp://ftp.oreilly.com/published/oreilly/nutshell/advanced_perl/examples.tar.gz
```

如果你没有浏览器的话,可以使用 Windows NT (或是 Windows 95) 下命令行方式的 FTP 客户端。

```
% ftp ftp.oreilly.com
Connected to ftp.oreilly.com.
220 ftp.oreilly.com FTP server (Version 6.34 Thu Oct 22 14:32:01 EDT 1992)
```

注6: 有几百种打印“Just Another Perl Hacker”的方式,大多数方法要归功于 Randal Schwartz。请见: <http://www.perl.com/CPAN/misc/japh>。

注7: 作为一名C代码复杂性大赛的裁判,我见多了那些拐弯抹角、晦涩难懂或者精彩纷呈的各种代码。你如果还不了解这个竞赛,就请访问<http://www.reality.sgi.com/~ioccc>。此外你如果觉得 Perl 还不够让人头脑发晕的话,就请去看一下位于<http://tpj.com/tpj.com/tpj/contest>的 Perl 复杂性大赛。

```
ready.
Name (ftp.oreilly.com:username): anonymous
331 Guest login ok, send e-mail address as password.
Password: username@hostname Use your username and host here
230 Guest login ok, access restrictions apply.
ftp> cd /published/oreilly/nutshell/advanced_perl
250 CWD command successful.
ftp> get README
200 PORT command successful.
150 Opening ASCII mode data connection for README (xxxx bytes) .
226 Transfer complete.
local: README remote: README
xxxx bytes received in xxx seconds (xxx Kbytes/s)
ftp> binary
200 Type set to I.
ftp> get examples.tar.gz
200 PORT command successful.
150 Opening BINARY mode data connection for examples.tar.gz (xxxx bytes) .
226 Transfer complete. local: examples.tar.gz remote: examples.tar.gz
xxxx bytes received in xxx seconds (xxx Kbytes/s)
ftp> quit
221 Goodbye.
%
```

## FTPMAIL

FTPMAIL 是任何可以通过因特网收发电子邮件的人都可用的一种邮件服务器。任何允许以电子邮件方式连通因特网的公司或服务提供商，都可以使用 FTPMAIL 这项服务。下面讲述如何来使用它。

你发送邮件给 `ftpmail@online.oreilly.com`。在内容栏内书写你想执行的 FTP 命令行。服务器将会替你执行匿名 FTP 命令，并将文件邮寄给你。要想获得完整的帮助文件，发送一封标题为空白，内容栏内只有“Help”一个单词的电子邮件。以下举例说明。下面的例子将会发送一个你选定目录下的文件列表和所要的例子程序文件。如果你对该例子程序的后续版本感兴趣的话，列表将非常有用。

```
Subject:
reply-to username@hostname (Message Body) Where you want files mailed
open
cd /published/oreilly/nutshell/advanced.perl
dir
```



```
get README
mode binary
uuencode
get examples.tar.gz
quit
```

可以在消息最后使用签名信息，但必须要放在“quit.”之后。

## 排版约定

本书中使用如下排版字体约定：

斜体 (*Italic*)

用于文件名和命令名称。也被用在电子邮件地址和 URL 上。

固定宽度 (Constant Width)

用于代码实例和代码元素名。

粗体 (**Bold**)

用于代码段，用以对关键部分的强调。也被用来表示用户的输入。

*Courier Italic*

用于强调代码段中的那些由工具自动生成的代码。

## 相关资源

有一些图书在我的职业生涯尤其是应用系统开发中极为有用，或许你也会同意我的观点。

1. 《Design Patterns: Elements of Reusable Object-Oriented Software》，Erich Gamma, Richard Helm, Ralph Johnson 和 John Vlissides. Addison-Wesley (1994)

2. 《Programming Pearls》, Jon Bentley. Addison-Wesley (1986)
3. 《More Programming Pearls》, Jon Bentley. Addison-Wesley (1990)
4. 《Design and Evolution of C++》, Bjarne Stroustrup. Addison-Wesley (1994)
5. 《The Mythical Man-Month》, Frederick P. Brooks. Addison-Wesley (1995)
6. 《Bring Design to Software》, Terry Winograd. Addison-Wesley (1996)
7. 《BUGS in Writing》, Lyn Dupre. Addison-Wesley (1995)

## Perl 资源列表

下面是有关 Perl 的书籍、杂志和万维网网站列表:

1. 《Programming Perl》第二版 (译注 5), Larry Wall, Tom Christiansen 和 Randal Schwartz. O'Reilly (1996)
2. 《Learning Perl》, Randal Schwartz. O'Reilly (1993)
3. 《The Perl Journal》, Jon Orwant 编辑, <http://www.tpj.com/>
4. Tom Christiansen 的 Perl 主页, <http://www.perl.com/perl/index.html>
5. Clay Irving 的 Perl 参考, <http://reference.perl.com/>

## 建议与评论

本书的内容都经过测试, 尽管我们做了最大的努力, 但错误和疏忽仍然是在所难免的。如果你发现有什么错误, 或者是对将来的版本有什么建议, 请通过下面的地址告诉我们:

美国:

O'Reilly & Associates, Inc.  
101 Morris Street

---

译注 5: 此书第三版已经出版。中文版即将由中国电力出版社出版。

Sebastopol, CA 95472

中国:

100080 北京市海淀区知春路 49 号希格玛公寓 B 座 809 室  
奥莱理软件(北京)有限公司

询问技术问题或对本书的评论, 请发电子邮件到:

[info@mail.oreilly.com.cn](mailto:info@mail.oreilly.com.cn)

我们为本书建立了一个网站, 以提供范例程序, 更正信息和本书未来版本的计划情况, 你可以通过如下地址访问:

<http://www.oreilly.com/catalog/advperl/>

最后, 您可以在 WWW 上找到我们:

<http://www.oreilly.com>

<http://www.oreilly.com.cn>

## 致谢

感谢我的妻子 Alka, 谢谢她使我在为该项目工作时, 免于承担日常的生活负担, 以及自从我们相识以来彼此之间无比快乐的好心情。

为我们所拥有的和所成就的一切, 感谢我们的父母。

感谢我的编辑, Andy Oram 和 Steve Talbott。他们孜孜不倦的对我的写作进行一次又一次的修订, 耐心的教我如何去写一本书。还有 O'Reilly 出版公司, 感谢它使作者和读者都能够愉快的贡献自己的一分力量。

感谢 Perl 的创始人 Larry Wall, 感谢他维护着这样一个友好方便的网上资源。感谢 Perl 5 Porters List 的长期贡献者们 (尤其是 Tom Christiansen), 感谢他们在

“业余”时间里对Perl的改进，书写相关的文档，毫无疲惫的宣传Perl。我对他们的精力和贡献羡慕万分。

感谢本书的技术审校，感谢他们万分仔细的阅读本书。Tom Christian, Jon Orwant, Mike Stok 和 James Lee 通读了整本内容并提供了极具洞察力的建议和鼓励。同时我还要向 Graham Barr, David Beazley, Peter Buckner, Tim Bunce, Wayne Caplinger, Rajappa Iyer, Jeff Okamoto, Gurusamy Sarathy, Peter Seibel 以及 Nathan Torkington 表示深深的谢意，感谢你们阅读本书的章节并提出了大量的宝贵建议。书中出现的任何错误和遗漏都是我造成的。衷心感谢非凡的引用语专家 Rao Akella，感谢你为本书所找到的恰当的引用语。

向提供了如此优异的工作环境的 Weblogic 和 TCSI 的同事们致谢。让我吃惊的是完成这么令人愉快的事，居然还能获得报酬。

感谢在 O'Reilly 为这本书工作的全体人员，感谢产品编辑 Jane Ellin，感谢 Mike Sierra 所提供的工具的支持，Robert Romano 的图片，Seth Maislin 的索引工作，Nicole Gipson Arigo, David Futato 和 Sheryl Avruch 的质量控制，Nancy Priest 和 Edie Freedman 的设计以及 Madeleine Newell 的产品支持。

最后向我所有的朋友表示感谢，感谢我们长时间一起散步、玩桌球的日子，还有你们鼓励的话语，以及当我沉迷于这本书的问题时你们的体谅与耐心。我真是幸福。





## 本章简介:

- 对已有变量的引用
- 使用引用
- 嵌套数据结构
- 对引用的查询
- 符号引用
- 引用的内部机制
- 其他语言中的引用
- 相关资源

# 第一章

# 数据引用与 匿名存储

如果我是玄虚的不可知论者的话，我就会搞不清  
自己到底是不是不可知论者，但是我不能肯定  
自己的感觉，因此我一定是一个玄而又玄的  
不可知论者（我猜是这样）。

——霍夫斯塔德《哥德尔，艾舍尔与巴赫》

主要有两方面的不同将用于创建真正复杂系统的编程语言，与普通的教学语言区分开来。那些更加健壮的编程语言具有以下特点：

- 不必使用变量名就能够动态地为数据分配存储空间。我们称之为匿名数据结构。
- 可以指向任何数据结构，无论它是静态分配还是动态分配的。

COBOL 语言在这方面是个例外；尽管它缺少这些特性，但还是获得了商业上的巨大成功。可这也是你为什么不会用它来开发飞行控制系统的原因。请看下面的句子描述的是一个相当简单的家谱。

23 岁的 Marge 与 24 岁的 John 结婚。

Jason, John 的兄弟，在 MIT 学习计算机科学。他刚刚 19 岁。

他们的父母 Robert 和 Mary，都是 60 岁，他们居住在佛罗里达。

Mary 和 Marge 的妈妈 Agnes，是童年时的小伙伴。

你此刻会在脑海中描绘出一张以圆圈代表人物,之间用箭头连接来表达关系的网络。试想一下你会如何用自己喜爱的编程语言来表述此类信息。如果你是一位C (或 Algol, Pascal, C++) 程序员,你或许会使用一种动态数据结构来存储每个人的个人数据 (如姓名, 年龄, 地点等), 然后用指针来表达人与人之间的关系。

指针, 简言之, 就是一个保存其他数据地址的变量。这个地址可以是机器地址, 在C语言中就是这样, 也可以是一种更高抽象层次的实体, 比如名字或数组偏移量。

C语言对这两方面的支持极其高效: 你使用 `malloc(3)` (注1) 来动态分配存储空间, 而使用指针来引用动态或静态分配的内存。虽然这样使用非常高效, 然而你却要花费大量的时间来和内存管理细节打交道, 认真仔细的创建和修改数据之间复杂的关系, 还要注意去除由于无效指针 (指那些所指向的内存已被释放或超出作用域的指针) 所引发的一系列致命错误。因此虽然写出来的程序效率会高一些, 而程序员都不会这样。

Perl 对这两种概念都支持的相当好。它允许你创建匿名数据结构, 而且还提供了一种称作“引用 (reference)”的类似于C语言指针的基本数据类型。正像C语言指针可以指向数据甚至函数地址一样, Perl的引用同样可以指向常规数据类型 (如标量变量、数组和散列表) 和其他类型的实体, 如子例程、`typeglob` 和文件句柄 (注2)。但和C语言不同, Perl的引用不允许用来操纵无类型的原始内存。

从提高程序员效率的角度来看, Perl 要更胜一筹。正如我们前面所见到的, 你可以使用更少的代码来创建复杂的数据结构。因为和C语言不同, Perl 为你省去了许多不必要的工作。比如下面这行代码:

```
$line[19] = "hello";
```

它创建了一个20个元素的数组, 然后将最后一个元素赋值为一个字符串。同样的工作在C语言中要花费好多行的代码。同等重要的是, 你不必再考虑内存分配问

---

注1: 圆括号中的数字在Unix中指代文档 (man 页面) 相应小节的约定。数字3表示描述C API 那个小节。

注2: 我们将在第三章“Typeglob 与符号表”中学习后面这部分的内容。



题。Perl 将确保一块数据当且仅当没有任何引用的情况下才被释放(这就确保了内存泄漏不会发生),而当有其他数据仍在指向它时不被释放(不会再有无效指针情况的发生)。

当然,仅仅因为有了这些,并不能使Perl成为实现航空调度系统一类复杂应用的首选语言。然而对于众多不是过分复杂的应用来说(不只是一次性的脚本程序),Perl 要比使用其他编程语言容易的多。

在本章你将学习以下内容:

- 如何创建对标量变量、数组和散列表的引用,如何利用引用来存取数据(间接访问)。
- 如何创建和引用匿名数据结构。
- Perl 如何使大家不用理会内存管理细节的内部机制。

## 对已有变量的引用

如果你有C语言编程背景的话(这对理解本章内容并不是必需的),就会知道有两种方法可以用来初始化C语言中的指针。你可以使用已有变量,如:

```
int a, *p;  
p = &a; /* p 中现在包含 a 的地址 */
```

此处的内存是静态分配的,也就是说是由编译器分配的。另外你也可以使用 `malloc(3)` 在运行中分配内存并获取指向它的地址:

```
p = malloc(sizeof(int));
```

这块动态分配的内存并没有名字(与那些拥有变量名的存储空间不同),它只能间接的通过指针来访问。这也是“匿名存储”这一称法产生的原因。

Perl 提供对静态和动态分配存储的引用。在这一节,我们要详细的学习前者。这样我们将能够针对引用和匿名存储这两个概念分别进行学习。

你可以通过在已有 Perl 变量名前添加 “\” 来创建对它的引用。例如：

```
# 创建若干变量
$a      = "mama mia"
@array  = (10, 20);
%hash   = ("laurel" => "hardy", "nick"=> "nora");

# 现在创建对它们的引用
$r_a    = \ $a;          # $r_a 现在引用 (“指向”) $a
$r_array = \@array;
$r_hash = \%hash;
```

同理，你也可以创建对常量的引用：

```
$r_a    = \10;
$r_s    = \"hello world";
```

既然数组和散列表是标量变量的集合，我们同样可以创建对其中单个元素的引用，只需添加前缀 “\”：

```
$r_array_element = \ $array[1];      # 引用标量变量 $array[1]

$r_hash_element  = \ $hash{"laurel"}; # 引用标量变量
                                           # $hash{"laurel"}
```

## 引用本身就是一种标量变量

引用变量，如 `$r_a` 或 `$r_array`，就是一种普通的标量变量——因此我们使用 “\$” 符。换句话说，标量变量可以是一个整数、一个字符串或者一个引用，而且还可以被重新任意的赋值为这些数据类型中的任一种。如果你将包含一个引用的标量变量打印出来，会显示如下结果：

```
SCALAR(0xb06c0)
```

虽然字符串和数字均有直接的打印表达形式，而引用却没有。于是乎 Perl 就打印出它能够理解的结果：所引用的值的数据类型及其内存地址。人们很少会用的着将引用打印出来，但是如果你非要将其打印出来的话，Perl 提供了一种默认的

输出格式。这也是Perl为何如此多才多艺的原因。“那么就不要再坐在那儿抱怨。只管去使用Perl，它不会让你失望的。”Perl对这句话可是当真的。

在我们讨论这个专题中，有一点很重要，那就是使用引用作为散列表的键值。Perl要求散列表的键值必须为字符串，因此当你将引用作为键值时，Perl则使用引用所代表的字符串（它将具有唯一性，因为它其实就是一个内存地址）。但当你而后从散列表中获得该键值时，它将仍旧是一个字符串，而不再被以引用的方式使用。也许未来版本的Perl会放宽这种只能以字符串作为散列表键值的限制，但是当前唯一能绕过这个问题的方法就是使用Tie::RefHash模块，这将在第九章中讨论。我要补充的是，这种限制在大一些的应用中微不足道。因为几乎很少有算法要求使用引用作为散列表键值，涉及这种限制的算法就更少了。

## 间接访问

间接访问（dereference）的意思就是取得引用所指的变量的值。

在C语言中，如果p是一个指针，那么\*p就表示p所指向的值。在Perl中如果\$r是一个引用，那么\$\$r，@r或%r分别是根据\$r所引用的数据类型（标量变量、数组、散列表）获取的相应的值。根据所引用的数据类型来选择使用正确的前缀是关键。如果\$r指向数组，那么你就必须得使用@r而不是%r和\$r。使用错误的前缀将导致运行时的严重错误。

让我们这么来考虑：在任何你要正常使用Perl变量（\$a，@b或%c）的地方，你均可以将该变量名字（a，b或c）替换为相应的引用变量（但要确保正确的引用类型）。任何使用普通数据类型的地方都可以用引用替代。

## 对标量变量的引用

以下是包含有标量变量的表达式，

```
$a += 2;
print $a;          # 正常打印出$a的内容
```

我们可以使用引用，这只需将“a”替换为“\$ra”：

```

$ra = \ $a;          # 创建对 $a 的引用
$$ra += 2;           # 注意不是 $a += 2;
print $$ra;          # 注意不是 $a

```

当然你必须确保 `$ra` 是一个指向标量变量的引用，否则 Perl 会提示运行时错误“Not a SCALAR reference (非标量变量引用)”。

## 对数组的引用

你可以用三种方式使用普通数组：

- 将数组作为一个整体进行存取，使用记号 `@array`。例如，你可以打印整个数组或向其中添加元素。
- 使用记号 `$array[$i]` 来存取数组中的单个元素。
- 使用记号 `@array[index1, index2, ...]` 来分段 (slice) 存取数组中的元素。

在以上三种情形下均可以使用引用。在以下例子代码中包含了各种情况，用以比较一般情况下使用数组和通过引用使用数组的不同：

```

$array = \@array;

push (@array, "a", 1, 2); # 将数组作为一个整体使用
push (@$array, "a", 1, 2); # 间接使用引用来访问数组

print $array[$i];        # 存取数组中的单个元素
print $$array[1];        # 将array替换为$array，间接使用引用
                        # 通过索引进行访问

@sl = @array[1,2,3];      # 以普通的数组分段方式访问数组
@sl = @$array[1,2,3];     # 通过引用来分段访问数组

```

注意在所有这些例子中，我们都是简单的通过把数组名替换为 `$array` 来间接的访问数组。

初学者老是搞不清数组变量和枚举列表(元素由逗号分隔)的不同，比如在枚举列表的前面插入“\”并不表示对枚举列表的引用：

```
$s = \('a', 'b', 'c',);      # 警告: 并不是你想要的结果
```

这和下面的代码结果相同:

```
$s = (\'a', \'b', \'c',);    # 对标量变量引用的列表
```

在标量变量上下文当中, 枚举列表总是返回最后一个元素, 也就是说 \$s 将包含常量字符串 c 的引用。在本节稍后部分, 我们可以利用匿名数组来解决这个问题。

## 对散列表的引用

对散列表的引用同样直接了当:

```
$rhash = \%hash;
print $hash("key1");      # 普通的散列表元素存取
print $$rhash{"key1"};    # hash 被替换为 $rhash
```

散列表分段存取同理也是这样:

```
@slice = @$rhash{'key1', 'key2'}; # 不是 @hash{'key1', 'key2'}
```

建议: 你必须抵制这种诱惑, 那就是尽管 Perl 提供了类似于指针的机制, 但是不要用它来实现一些诸如链表和树这样的基本数据结构。对于少量的数据元素, 标准的数组类型就能够提供相当不错的插入与删除性能, 而且其资源消耗相比用 Perl 原语编写的链表要低得多。(在我的机器上, 一项简单的测试, 向一个 Perl 数组的头部插入大约 1250 个元素, 比在用 Perl 创建链表上进行同样的操作要快。)同样如果你想使用 B 树的话, 你应该先去了解一下 Berkeley DB 库 (将在第十章“持续性”中讨论), 而后再决定是否使用 Perl 语言所编写的替代品。

## 优先级

那些包含有键值的表达式也许会把人搞的有些糊涂。你将 \$\$rarray[1] 理解为 \${\$rarray[1]} 还是 { \$\$rarray } [1] 或者是 \${\$rarray} [1] ?

(请稍微多看会儿, 看仔细!)

事实上，最后一个是正确的。Perl 解析此类表达式遵循两个简单的规则：(1)键值和索引定位被放到最后，(2)离变量名最近的前缀最先联编。每当 Perl 看到 `$$rarray[1]` 或 `$$rhash{"browns"}`，它会将索引定位（`[1]` 和 `{"browns"}`）放到最后进行。这就只剩下 `$$rarray` 和 `$$rhash`。它将给予离变量名最近的 ‘\$’ 以最高的优先权。因此就产生如下结果：`${$rarray}` 和 `${$rhash}`。将第二条规则视觉化的另一种方法就是优先级从右到左递减（变量总是在一系列符号的最右端）。

要注意的是，我们不是在谈论操作符的优先级，因为 `$`、`@` 和 `%` 不是操作符。以上规则表示表达式如何被 Perl 解析。

## 简明的箭头记号

Perl 提供了另一种简单易读的用以存取数组和散列表元素的语法结构：“`->[]`”记号。例如，如果给出了数组的引用，你可以这样来获取数组中的第二个元素：

```
$rarray = \@array;  
print $rarray->[1] ;    # 易读，简明的方式
```

而以前的访问方式为：

```
print $$rarray[1];      # 杂乱，还得考虑优先级  
print ${$rarray}[1];  # 打了“绷带”的方式！
```

我更倾向于使用箭头记号，因为它视觉上更干净些。

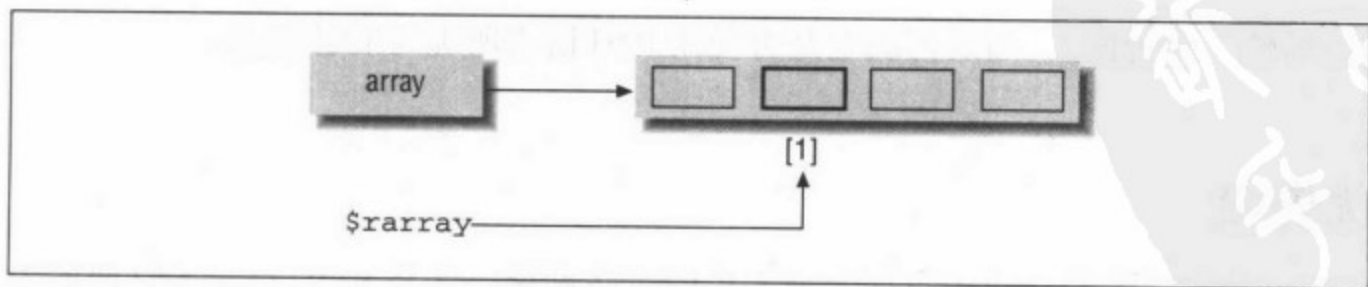


图 1-1 `$rarray ->[1]` 图解

与数组类似，你可以通过使用 `->{ }` 记号来存取散列表中的数据元素：

```
$rhash = \%hash;
print $rhash->{"k1"};

# 也可以使用.....
print $$rhash{"k1"};
# 或者使用
print ${$rhash}{"k1"};
```

注意：这种记号仅仅对单键值的索引起作用，而对于分段存取则不行。考虑以下例子：

```
print $rarray->[0, 2]; # 警告:这里并不表示间接分段存取数组
```

Perl 将把括号中的内容视为由逗号分隔的表达式，该表达式将返回最后一个元素所指的数组元素：2。因此上面的表达式也就等同于 `$rarray->[2]`，这是一个单一索引存取而不是分段存取。（回想一下先前提到的规则：一个枚举或逗号分隔的列表，在标量变量上下文中，总是返回最后一个元素。）

## 不存在自动间接访问

Perl 不会进行自动间接访问（注3）。你必须使用刚才讲到的结构来执行显式的间接访问操作。这和 C 语言一样，在 C 语言中也必须使用 `*p` 来表示 `p` 所指的对象。考虑以下例子：

```
$rarray = \@array;
push ($rarray, 1, 2, 3); # 错：$rarray 是一个标量变量，而不是数组
push (@$rarray, 1, 2, 3); # 对
```

`push` 操作要求其第一个参数为数组而不是指向数组的引用（而引用则是一个标量变量）。同理，在打印数组时，Perl 不会自动间接访问任何引用。例如：

```
print "$rarray, $rhash";
```

这将打印出如下结果：

---

注3：正如我们将要在第三章见到的，文件句柄除外。



```
ARRAY(0xc70858), HASH(0xb75ce8)
```

这似乎算不上是个问题，但是在两种情况下会导致十分晦涩难懂的结果。第一种情况是在算术表达式或条件表达式中错误地使用引用，例如，如果你将 `$a += $$r` 错误的写成了 `$a += $r`，那么这个错误将是很难查到的。第二种常见的错误是将数组 (`$a = @array`) 而不是数组的引用 (`$a = \@array`) 赋值给标量变量。任何一种情况下，Perl 均不会显示警告信息。此处适用 Murphy 定律：“问题只有在你向顾客作演示时才被发现。”

## 使用引用

要想创造复杂的数据结构，引用是必不可少的。因为下一章中主要的内容就是讲述这个话题，所以我们在此不多费口舌。下面的章节列举了 Perl 对间接性和存储管理支持的其他优点。

## 向子例程传递数组和散列表

当你要向子例程中传递一个以上的数组或散列表时，Perl 将会把它们合并到 `@_` 数组中供子例程使用。唯一避免合并的方法就是传递输入数组或散列表的引用。下面是一个将一个数组中的元素与另一个数组中对应元素相加的例子：

```
@array1 = (1, 2, 3); @array2 = (4, 5, 6, 7);
AddArrays (\@array1, \@array2); #以引用方式传递数组
print "@array1 \n";
sub AddArrays
{
    my ($rarray1, $rarray2) = @_;
    $len2 = @$rarray2;    #array2 的长度
    for ($I = 0 ; $I<$len2 ; $I++) {
        $rarray1->[$I] += $rarray2->[$I];
    }
}
```

在这个例子中，我们向 `AddArrays` 传递了两个数组引用，该子例程对它们间接访问以后，计算出数组的长度，并将对应元素对加。

## 运行效率

通过使用引用，我们可以高效的向子例程中传入或传出大量数据。

然而，传递标量变量的引用一般在性能上决不是最优的。我经常看到这样的例子，程序员在从文件中一行行读数据时，为了把拷贝操作降到最少，常这么来写：

```
while ($ref_line = GetNextLine()) {  
    .....  
    .....  
}  
sub GetNextLine () {  
    my $line = <F>;  
    exit(0) unless defined($line);  
    .....  
    return \ $line;          # 为避免拷贝操作，以引用的方式返回  
}
```

GetNextLine 为了避免拷贝操作，将所读取的行以引用的方式返回。

你或许会很吃惊，这种策略竟然对于提高应用的整体性能微不足道。因为大多数时间都用在读文件数据并操纵 \$line 上面了。而同时，使用 GetNextLine 就必须使用一种间接 (\$\$ref\_line) 的方式，而不是更为直接了当的 \$line (注4)。

附带说一句，你可以使用标准库中被称为基准测试 (Benchmark) 的模块来比较一下两种不同代码实现方式所花费的时间，如下所示：

```
use Benchmark;  
timethis(100, "GetNextLine()"); # 执行 GetNextLine 100 次并计算
```

所花费的时间在该模块中定义了一个称作 timethis 的子例程，它以一段代码为参数并执行你所设定的次数，然后打印出花费的时间。我们将在第六章“模块”中讲解 use 语句。

---

注4： 这里是指典型的情况。大多数应用处理的行是 60~70 个字节长。

## 匿名存储的引用

到目前为止，我们学习创建了对已存在变量的引用，现在我们要学习对“匿名”数据结构的引用，也就是那些没有同变量名关联的值。

创建匿名数组，需要使用方括号而不是圆括号，如：

```
$ra = [ ];          # 创建一个空的匿名数组，并返回对它的引用
$ra = [1, "hello"]; # 创建一个经过初始化的匿名数组，并返回对它的引用
```

这种表达形式不仅分配了匿名存储空间，还返回了对它的引用，非常类似于C语言中的 `malloc(3)`，它返回的是一个指针。

如果你使用圆括号而不是方括号会是什么结果？再次回想一下，Perl 将把等号右面的部分看作以逗号分隔的表达式，计算并返回最后一个元素的值；`$ra` 将赋值为“hello”，这或许不是你想要的结果。

要创建匿名散列表，就用大括号代替方括号即可：

```
$rh = {};          # 创建一个空的散列表并返回对它的引用
$rh = {"k1", "v1", "k2", "v2"}; # 一个经过初始化的匿名散列表
```

所有这两种记号都很容易记忆，因为它们与其相应的数据类型使用相同的种类的括号——数组用数组类型的括号，散列表用散列表类型的括号。你可以和通常情况下创建有名散列表的方法对比一下：

```
# 这是一个普通的使用 % 前缀的散列表，它由圆括号所包围的列表初始化
%hash = ("flock" => "birds", "pride" => "lions" );

# 匿名散列表是一组用大括号括起来的列表
# 该表达式的结果是一个标量变量，值为指向散列表的引用
$hash = { "flock" => "birds", "pride" => "lions" };
```

那么存在动态分配的标量变量吗？其实 Perl 并不提供完成此类工作的任何记号。或许你根本就用不着它。如果你非得寻求一种途径的话，可以使用下面的技巧：首先创建一个指向现存变量的引用，然后让那个变量超出作用域。

```
{
    my $a= "hello world";  #1
    $ra = \"$a;            #2
}
print "$$ra \n";          #3
```

“my”操作符可以将一个变量置为私有变量（用Perl的行话就是，局部化）。你也可以使用操作符“local”，但是它们之间有一种微妙却十分重要的差异，这一点我们将在第三章里阐述。对于这个例子，任何一种操作符都行。

这样，\$ra就是一个指向局部变量\$a的全局变量。\$a通常在代码块的末尾被删除，但是由于\$ra仍然引用它，于是为\$a分配的内存并没有被释放。当然，如果你又将\$ra赋值为别的值的话，这块空间将在\$ra被赋给新值前释放。

你可以如下创建常数型简单数据变量：

```
$r = \10;  $rs = \"hello";
```

常数是静态分配和匿名的。

引用变量并不关心自己所指的是匿名数值还是已存在的变量。这同C语言中的指针行为一样。

## 多重引用的间接访问

我们已经看到了引用是如何指向包括其他引用在内的其他实体了（这些引用本身就是普通标量变量）。也就是说我们可以像下面这样创建多重引用：

```
$a      = 10;
$ra     = \"$a;          # 引用指向$a的值
$rra    = \"$ra;         # 指向引用$a值的引用的引用
$rrra   = \"$rra;        # 引用的引用的引用……
```

现在我们要对这些引用间接访问。下面所有的语句都将产生同样的数值（也就是\$a的值）：

```

print $a;           # 打印结果为 10, 下面的语句将打印同样的结果
print $$ra;         # 对 $a 的一次引用
print $$$rra;       # 把 ra 替换为{$rra}, 仍为一层指向 $a 数值的引用
print $$$$rrra;     # .....依次类推

```

巧的是,上面这个例子中演示了一种微软 Windows 编程人员所熟悉的惯例,那就是“匈牙利记号”(注5)。每个变量名都根据其类型加上前缀(“r”代表引用,“rh”表示对散列表的引用,“i”表示整数,“d”表示双精度浮点数等等)。下面的这种写法立刻就可以看出存在问题:

```

$rh_collections[0] = 10;      # 警告: 'rh' 被用作数组了

```

你有一个变量,名为 \$rh\_collections。根据命名习惯(前缀 rh)这或许是一个指向散列表的引用。但你却把它当作指向数组的引用来使。当然 Perl 会产生运行时例外来警告你(“第2行不是指向数组的引用”)。你在写程序时通过检查程序代码就可以发现此类错误,而且要比在代码测试阶段花大力气覆盖所有代码路径来找出运行时错误容易的多。

## 一种更加通用的规则

前面早些时候,在讨论优先级时,我们讲过 \${\$rarray}[1] 实际上和 \${\$rarray}{1} 是等价的。我们选择花括号来加以划分并不完全是偶然的。因为这里存在一种更为通用的规则。

花括号表明了一个代码块,而且并不关心你在里面都放些什么,只要其最终返回指向所要求数据类型的引用即可。像 \${\$rarray} 这样简单的表达式显而易见返回的是一个引用。对照一下下面的例子,它在括号中调用了返回引用的子例程:

```

sub test {
    return \ $a;      # 返回一个指向标量变量的引用
}

```

---

注5: 当 Charles Simonyi 在微软开始使用这项约定之后,它在因特网上就是一种火爆的争论话题;有人喜欢有人厌恶。显然,即使是在微软,系统开发人员使用它而应用开发人员则不用。在像 Perl 这样一种没有强制类型检查的语言中,我推荐在方便的话就使用它。

```
}
$a = 10;
$b = ${test()};      # 在代码块中调用 test 子例程，该子例程返回指向 $a 的引用
                      # 该引用被间接访问
print $b;             # 打印结果为 "10"
```

总结一下，一个返回引用的块可以放置在任何变量名可以出现的地方。例如，除了 \$a 我们还可以使用 \${\$ra} 或 \${\$array[1]} (前提是 \$array[1] 为指向 \$a 的引用)。

大家还记得块中可以放置任意数量的语句，而且最后一条语句的返回值代表该块的值。除非你想成为“Perl 复杂性大赛”当真的争夺者，否则就不要在使用这条通用的间接访问规则时，书写内容超过两行的代码块。

## 特洛伊木马

在我们谈论混乱的问题时，值得提一下一种阴险的在字符串中隐藏可执行代码的方式。正常情况下，当 Perl 看到字符串如 \$a 时，它将进行变量替换。但是你现在知道了，a 可以被返回指向标量变量的代码块所替换，那么下面的这条语句即便被包裹在字符串中也完全是合法的：

```
print "${foo()}";
```

我们现在将 foo() 替换为 system ('/bin/rm \*')，这可就成了一个令人不快的特洛伊木马了。

```
print "${system('/bin/rm *')}"
```

Perl 像对待其他别的函数一样，相信 system 会返回一个指向标量变量的引用。在 Perl 确认 system 并没有返回指向标量变量的引用时，破坏已经造成了。

经验：一定要留神你从不信任的数据源获取的字符串。使用污染模式 (taint-mode) 选项 (这样调用 Perl 解释器：perl -T) 或 Perl 发行版中的 Safe 模块。有关污染检测的内容请看 Perl 的文档，有关 Safe 模块的信息可以查阅文档索引中的相关连接。

## 嵌套数据结构

大家知道数组和散列表只能包含标量变量类型的元素；它们不能直接包含别的数组或散列表。但是引用可以指向数组和散列表，而且引用本身就是标量变量，现在你或许会知道数组或散列表中的元素是如何指向其他的数组或散列表了。在这一节，我们将学习如何创建嵌套的异构数据结构。

举个例子，我们要记载某个人的详细情况和其被抚养者的详细情况。我们采取为每个人创建独立的有名散列表的方式：

```
%sue = (                                # 家长
    'name' => 'Sue',
    'age'  => '45');
%john = (                                # 孩子
    'name' => 'John',
    'age'  => '20');
%peggy = (                                # 孩子
    'name' => 'Peggy',
    'age'  => '16');
```

可以按如下方式将 John 和 Peggy 的数据结构同 Sue 的关联起来：

```
@children = (%john, %peggy);
$sue{'children'} = \@children;
# 或者这么写
$sue{'children'} = [%john, %peggy];
```

图 1-2 描述了这样建立起来的数据结构。

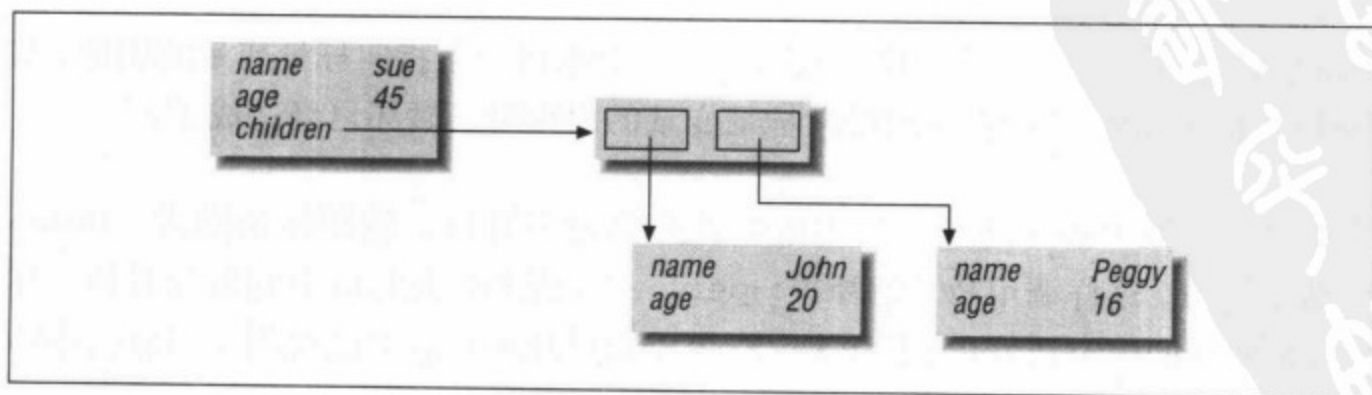


图 1-2 混合使用标量变量和数组与散列表



给出 %sue, 你可以如下这样打印出 Peggy 的年龄:

```
print $sue{children}->[1]->{age};
```

## 隐含的创建复杂的数据结构

假设以下语句是你程序的第一行:

```
$sue{children}->[1]->{age} = 10;
```

Perl 将会自动的创建散列表 %sue, 使其拥有一个以 children 为键值的散列元素, 并让它指向一个新建的数组, 而它的第二个元素被设置为指向一个新建的散列表, 该表则拥有一个以字符串 age 为键值的元素。这就是程序员的效率。

## 最后的缩写: 省去两个下标间的箭头

谈到程序员的效率, 让我们谈论一个节省键盘敲击次数的技巧。在两个下标之间 (也只有在下标之间) 你可以省去箭头符号 “->”。也就是说, 下面的这两条语句是等效的:

```
print $sue{children}->[1]->{age};
print $sue{children}[1]{age};
```

这同 C 语言中实现多维数组有点相似。在 C 中除最后一个外, 每个索引都像指针一样指向下一个层次 (或称之为维)。只有最后的索引才与实际的数据关联。C 与 Perl 的方式还多少有些差异, C 语言中将  $n$  维数组看成是连续的字节流, 并不为指针和子数组分配空间。而 Perl 则为其指向下一级一维数组的引用分配空间。

接着我们前面没讲完的话题讲, 你会看到即便是如此简单的例子, 也更多的得益于匿名的数组和散列表, 而不是有名的数组和散列表:

```
%sue = (
    'name'      => 'Sue',
    'age'       => '45',
    'children' => [
                                # 家长
                                # 拥有两个散列表的匿名数组
                                # 匿名散列表 1
    ]
);
```

```
        'name' => 'John',  
        'age'  => '20'  
    },  
    {                                     # 匿名散列表 2  
        'name' => 'Peggy',  
        'age'  => '16'  
    }  
]  
};
```

这段代码只有一个有名变量。子属性是一个指向匿名数组的引用，而数组的元素则又包含了指向带有孩子详细信息的匿名散列表。这种嵌套根据需要可以有任意层；比如，你可以将 John 的受教育情况作为一个指向匿名数组的引用来表示，而该数组中的元素又可以指向散列表（每个散列表又包含诸如学校、学分等资料）。这些数组或散列表实际上均没有把下一级的数组或散列表嵌入其中；要知道匿名数组或散列表的语法要求的是引用，也就是说，这些包含结构所看到的只是些引用。换句话说，这种嵌入形式并不表示一种包含层次。试着执行一下 `print values(%sue)` 就知道了。

好在当顶层数据结构（`%sue`）被删除，或重新被赋值为其他数值时，所有嵌套的数据结构将被自动删除。仍被其他地方引用的内部数据结构或元素则不会被删除。

## 引用的查询

`ref` 函数被用来查询标量变量是否包含一个引用，是的话，再判定所指向的是什么数据类型。如果 `ref` 的参数包含的是数字或字符串，就返回 `false`（这是一个布尔值，不是字符串），若是引用，就会返回代表所指向的数据类型的字符串：“`CALAR`”，“`HASH`”，“`ARRAY`”，“`REF`”（它表示指向另一个引用变量），“`GLOB`”（指向 `typeglob` 类型），“`CODE`”（指向子程序），或“`package name`”（表示对象属于这个包——我们将在以后详细讲解）。

```
$a = 10;  
$ra = \ $a;
```

`ref($a)` 将返回 `FALSE`，因为 `$a` 不是引用。

`ref($ra)`将返回“SCALAR”字符串，因为`$ra`指向一个标量变量。

## 符号引用

一般来说，类似`$$var`的结构表示，`$var`是一个引用变量，而且程序员希望该表达式能够返回`$var`所指向的值。

假如`$var`不是引用变量的话会出现什么情况呢？Perl并不是断然打印出错误信息，而是尝试检查`$var`的值是否为一字符串。如果是，Perl将以该字符串作为正规的变量名与这个变量重新加以组合！考虑下面的例子：

```
$x = 10;
$var = "x";
$$var = 30;          # $x的值将被改为30，因为$var是一种符号引用
```

当对`$$var`进行间接访问时，Perl首先检查`$var`是否是引用，在这里它不是，而是字符串。于是Perl会再给该表达式一次机会：它将`$var`的内容当作变量标识符（`$x`）。这样一来，最终`$x`的值就被修改成30了。

应当注意重要的一点是符号引用只对全局变量有效，而不能应用于那些用`my`标识为私有的变量。

符号引用对数组和散列表同样适用：

```
$var = "x";
@$var = (1, 2, 3);      # 将@x的值设置为右边的枚举列表
```

注意`$var`前面的符号指示变量的存取类型：`$$var`等价于`$x`，`@$var`也等价于`@x`，这种特性非常有用。

对于那些在Perl的早期版本中就实现过此类功能的人来说，这要比使用`eval`更为高效。举例说明，你要在自己的脚本程序中，处理诸如“-Ddebug\_level=3”的选项，并设置`$debug_level`变量。下面是其中的一种实现方式：

```
while ($arg = shift @ARGV){
```

```
if ($arg =~ /^-D(\w+)=(\w+)/) {  
    $var_name = $1; $value = $2;  
    $$var_name = $value;      # 更为简洁可书写为: $$1=$2;  
}  
}
```

从另一方面来说, Perl 竭尽全力使一个表达式正常工作, 有时却于事无补。在前面的例子中, 如果你所希望程序使用一个真正的引用, 而不是什么字符串的话, 那么你就会希望 Perl 能够指出这种情况而不是做任何的假定。幸运的是, 这种“急于求成”的方式可以被关闭。Perl 有许多编译时指令和指示。strict 编译指示将告诉 Perl 做严格的错误检查。你甚至可以指定需要进行严格检查的范围, 其中之一就包括“refs”这一项:

```
use strict 'refs'; # 告知 Perl 不允许符号引用  
$var = "x";  
$$var = 30;
```

无论何时你试图使用符号引用时均会导致运行时错误:

```
can't use string ("x") as a SCALAR ref while "strict refs" in use at  
try.pl line 3
```

strict 指令直至程序块结束依然有效。可以通过输入 no strict 或更明确的 no strict 'refs' 将其关闭。

## 内部工作细节

现在我们来查看 Perl 在内部是如何管理内存的。你可以跳过本节, 不会存在连贯性的问题。

正如图 1-3 所示变量在逻辑上表示为名字与数值之间的联编 (注 6)。

---

注 6: 无论变量是全局的、动态作用域的还是词法作用域的 (使用 my) 均是如此。第三章中有更为详细的描述。

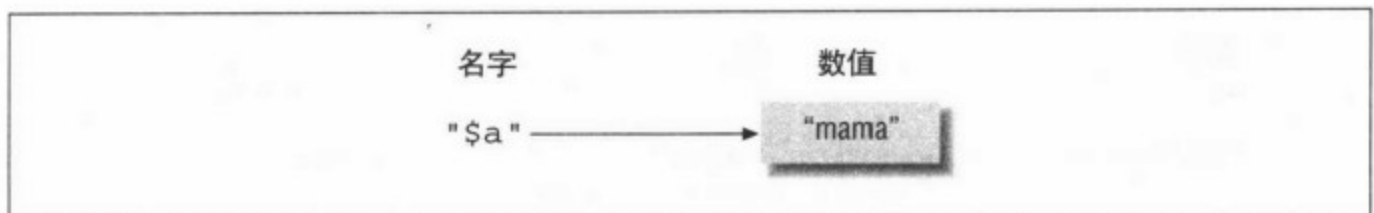


图 1-3 变量是一种名字和数值对

数组或散列表并不仅仅是数字或字符串的集合。它是标量变量值的集合，这种区别很重要，如图 1-4 所示。

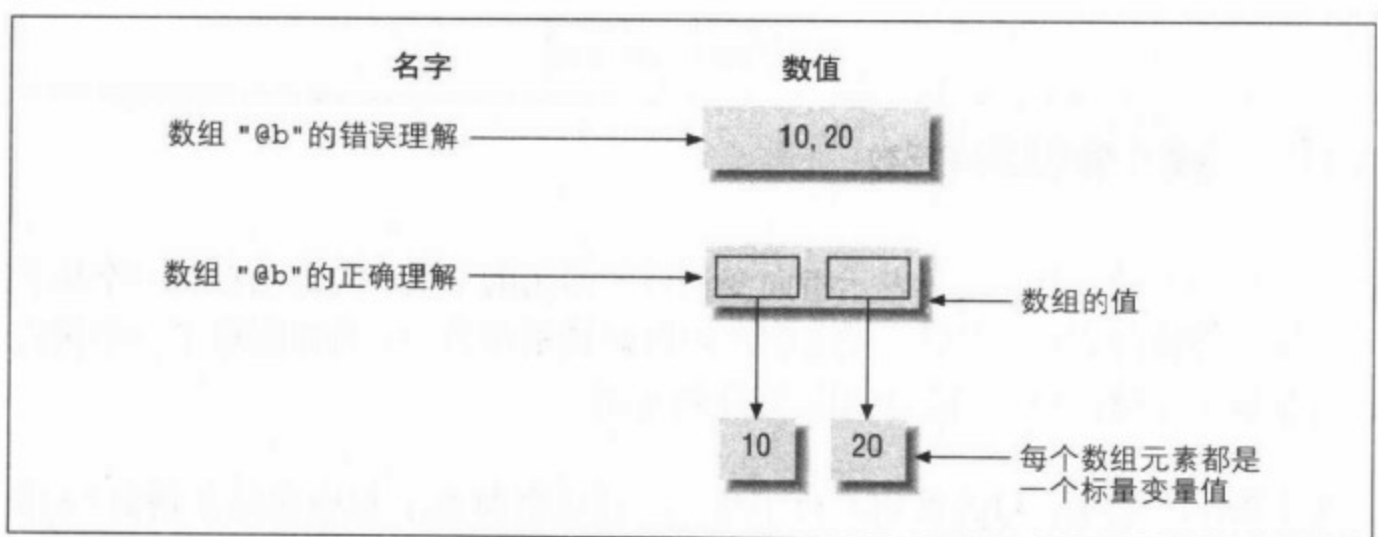


图 1-4 数组的值是标量变量值的集合

图 1-4 中每一个方框代表一个不同的值。一个数组拥有一个代表所有标量变量值集合的值。数组中每个元素又有一个标量变量值。这类似于将一群狮子当作一个整体来看待（这也是我们使用单数形式的原因），拥有与单个狮子不同的整体属性。

值得注意的是尽管名字总是指向数值，而数值并不一定必须要求要有名字来指向它，正如我们在图 1-4 中的数组元素，或匿名数组和散列表的举例说明中看到的。

## 引用记数

为了提供简单和透明的内存管理，无论是否有名字，Perl 为每个数值保留一个引用记数 (reference count)。让我们在前面的图例中增上这一内容，请参考图 1-5。

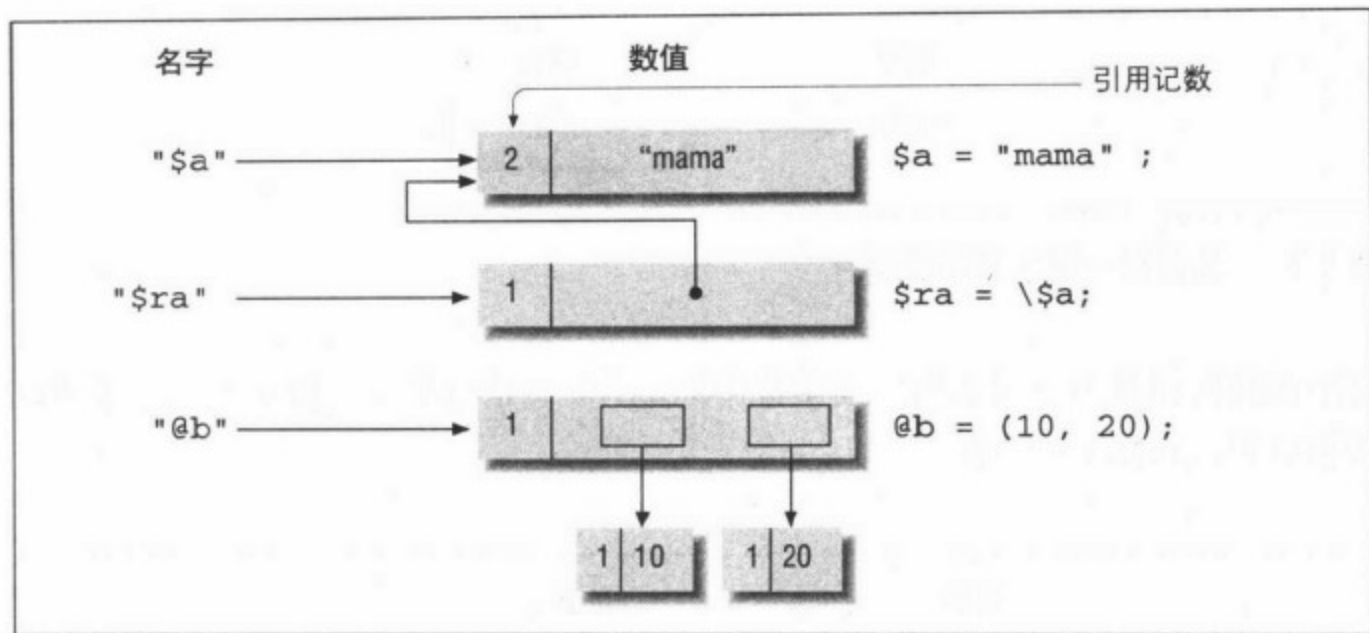


图 1-5 为每个值加上引用记数

正如你所看到的，引用记数表示指向变量值的箭头的个数。因为总存在一个从名字指向其数值的箭头，所以变量的引用记数应该至少为一。当你获取了一个指向该变量的引用时，相应值的引用记数将被递增。

需要强调的一点是，尽管我们习惯于将 `$ra` 看成指向 `$a`，但它实际上指向 `$a` 的值，`$ra` 甚至不知道自己指向的值是否在符号表中有对应项。引用变量的值是另一个标量变量值的地址，即使是 `$a` 的值变了，其地址也不会改变。

Perl 将自动删除那些引用记数为零的值。当变量（有名的值）超出作用域时，名字与值之间的连接被自动删除，值的引用技术递减。在通常引用记数为一的情况下，该值将被删去（或称作垃圾收集，garbage collected）（注 7）。

这种引用记数技术有时被称为“穷人的垃圾收集”法，用以对比在诸如 LISP、Java 和 Smalltalk（尽管早期版本的 Smalltalk 使用引用记数）中使用的更为复杂的技术。问题就在于引用记数需要耗费空间。考虑到你的应用中的每个数据都要一个额外的整数与之关联，这加起来并不是一个小数目。

而且还存在循环引用的问题。举一个最简单的例子：

注 7： 为了保证效率，Perl 并不真正删除它；而只是将它发送到自己的空闲缓冲池中，并在你需要一个新值时重用它。然而，它在逻辑上是被删除了。

```
$a = \$a;
```

这是一个经典的自我主义的例子。`$a`的引用记数表明存在指向它的东西，因此它将永远不被释放。更为实用的循环引用例子就是网络图（每个节点都知道自己每个相邻节点），还有环型缓冲区（尾部元素指向首部元素）。在Java和Smalltalk中实现的现代化的垃圾收集算法，能够检测出这种循环引用的存在，并在不存在任何变量对其中元素的引用时，释放该结构。

从另外的角度来说，引用记数简单而易于理解和实现，可以使Perl语言很容易的同C/C++代码集成。请参考本章最后“资源”一节有关垃圾收集技巧的详细论述。

要注意的是，尽管符号引用允许你间接的访问任何变量，但是并没有实际的创建引用变量。也就是说通过符号引用方式存取的变量，其引用计数不会改变。因此符号引用常被称作“软”引用，与“硬”引用（它们为这种间接操作分配实际的存储空间）相反。

这和Unix文件系统中软连接和硬连接的概念有点类似。每当有人创建对一个文件的硬连接时，该文件的引用计数就会递增。因此，除非文件的引用计数递减为零，否则该文件的内容并不能被真正删除。而符号连接只保存有文件名，而且可以指向一个并不存在的文件；只有通过符号连接去打开这个文件时才会发现这种情况。

## 数组 / 散列表引用与元素引用的对比

我们知道，数组作为一个整体，同组成它的标量变量值元素是有差别的。数组的值单独维护自己的引用计数，而其组成元素也都有它们自己的引用计数。当你创建对数组的引用时，其引用计数就会递增，而那些组成元素的引用计数则不受影响。如图1-6所示。

作为对照，图1-7描述了你创建对数组或散列表中元素的引用时的情形。

当你创建对数组（或散列表）中元素的引用时，Perl会使该标量变量值的引用计数增加。比如说，你执行了`pop`操作，它的引用计数就会减1，因为数组不再对该标量变量值感兴趣了。但是由于仍然存在一个对该数组元素的引用（而且它的引用计数仍然为1），所以它不会被释放。图1-8描述了对`@array`执行一次`pop`操作后的情形。



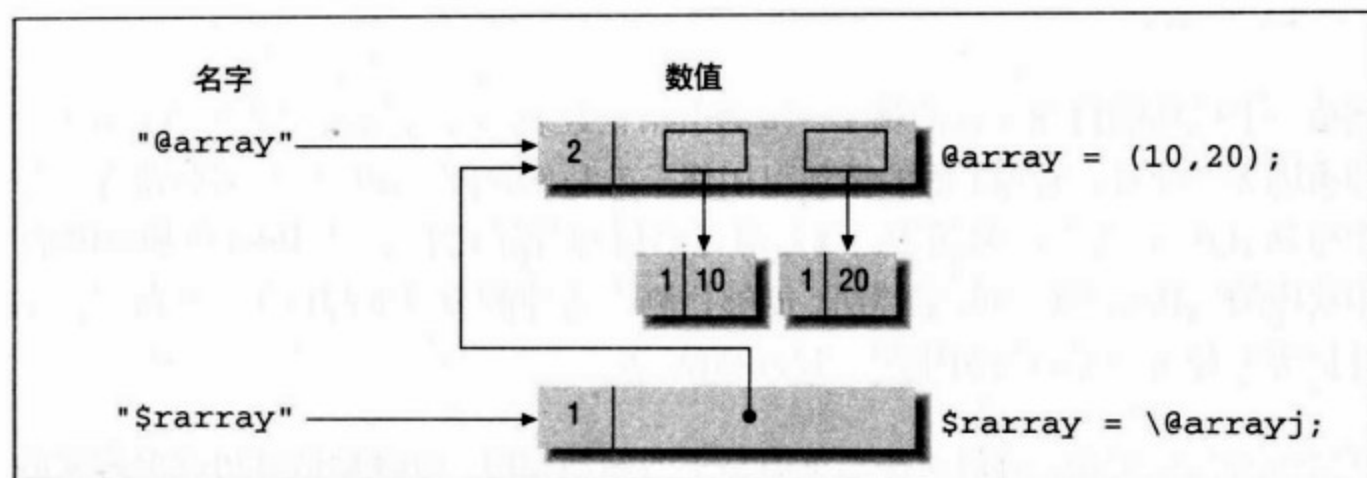


图 1-6 创建对数组的引用

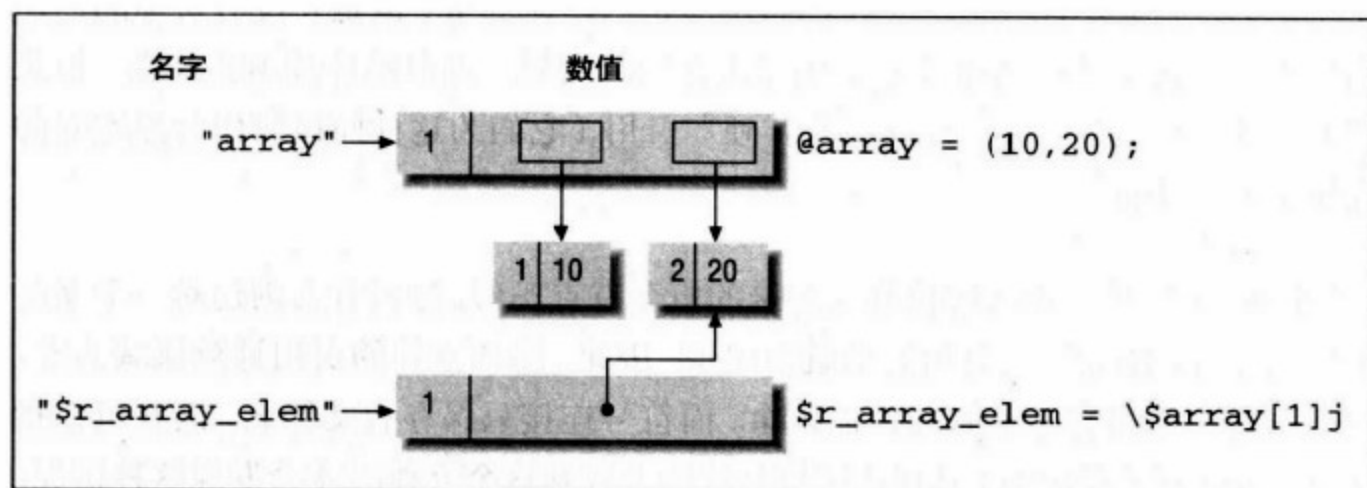


图 1-7 引用表元素

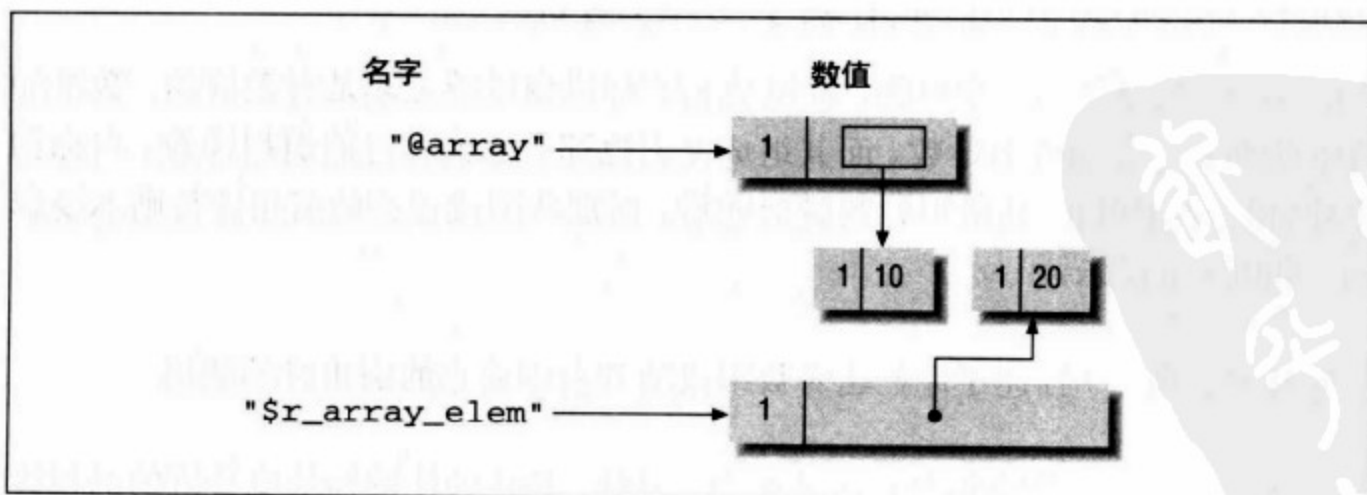


图 1-8 对 @array 执行一次 pop 操作; \$r\_array\_elem 仍然保留对已弹出的标量变量的引用



## 其他语言中的引用

### Tcl

Tcl 不提供动态创建匿名数据结构的功能。但是作为一种动态语言，它支持运行时创建新的变量（自动的为变量提供名字）。这种方式不但速度慢而且还容易出错。而且，唯一以引用形式来传递变量的方法，就是通过传递变量的名字，这与 Perl 中的符号引用是等效的。所有这些使 Tcl 语言很难创建复杂的数据结构（而且即便是作出来了也难以维护）。但是从整体上公平的讲，Tcl 重点是一种用来连接各类应用与工具包的胶合语言，大多数复杂的处理都应在基于 C 的应用中完成，而不是把它放在脚本程序中去。Tcl 从开始就不想设计成全能型的脚本开发环境。（可是我还是听说，它的局限性并没有妨碍人们用它去开发驱动石油钻塔的长达 50000 行的脚本程序。）

### Python

Python 除了在基础数据类型的处理上与 Java 不同外，其他方面都很相似。所有的对象都以引用形式传递。就是说如果将一个具有链表值的变量赋值给另一个变量，就会使第二个表变量成为前者的别名。如果你需要的是一个拷贝的话，你必须采用显示的操作，而且还要付出相应的性能损失。相比 Perl 语言，我更喜欢 Python，因为我们更需要对数据结构的引用而不是拷贝副本。而这种简单，高效的默认存取方式很好。

和 Perl 一样，Python 支持每一种数据类型的引用计数，包括在 C/C++ 扩展中定义的用户数据类型在内。

### C/C++

C/C++ 支持可以在编译时进行检测的类型安全指针。因为指针包含了数据的原始地址，所以对一块数据的引用简洁而高效。从另一个角度讲，程序员要担负起所有内存管理的责任。查看 Tcl、Perl 和 Python 这些解释器（都是用 C 实现的）的内存管理策略，会很有意义。

C++ 支持引用的概念，你可以通过它来创建现存变量的别名。该特性让人想起 `typedef` 的别名机制（这将在第三章中学到），而它又和 Perl 的引用决不相同。

## Java

在 Java 语言中，除基础数据类型如 `int` 和 `float` 外的一切，均以引用方式传递。由于 Java 基础架构就支持垃圾收集（它单独运行在一条单独的线程中，所以不会阻塞应用系统线程），所以不存在内存管理问题。Java 拥有与 C++ 一样丰富的数据类型，而且没有内存管理的烦扰，因此它对大型的应用编程来说很有吸引力。

## 相关资源

1. Perlref (Perl 的技术文档)。
2. Uniprocessor Garbage Collection Techniques. Paul Wilson. International Workshop on Memory Management, 1992。

这篇文章详细阐述了有关垃圾收集的问题。获取地址为：

<ftp://ftp.cs.utexas.edu/pub/garbage/gcsurvey.ps>。



本章简介:

- 用户定义的结构
- 有关矩阵的例子
- 教授、学生与课程
- 颁奖
- 漂亮的打印输出
- 相关资源

## 第二章

# 实现复杂的数据结构

蜘蛛们，不要担心，我偶尔才打扫一下房间。

—— Kobayashi Issa

Perl 的成功说明了这样一个事实，许多问题仅使用基本的数据类型就可以解决。Jon Bentley 在所著的《Programming Pearls》和《More Programming Pearls》两本书中则更深入阐明了，如果基本数据结构是动态的而且内存管理自动进行的时候，解决问题的能力会是多么强大。但是当从脚本到应用系统领域的程序变得越来越复杂时，这种用更加复杂的方式来表示数据的需求日益迫切，超出了有时单靠基本数据类型就能实现的极限。

本章我们将在几个“真实”的例子中运用第一章学习的语法和概念。我们将编写代码来为文件中的数据创建复杂的数据结构，并灵活使用令人眼花缭乱的\$和@符号。对于每个问题，我们将以不同的方式表达相同的数据，并学习如何权衡程序的效率与程序员的效率。为了清楚起见，我们将不在错误处理上花费太多的精力。

Tom Christiansen 曾写了一系列被称之为 FMTEYEWTK (“Far More Than Everything You've Ever Wanted to Know!”，即“比你想知道的任何事都多!”) 的指导教程。这套在 Perl 新闻组中出现的教程涉及各类话题。它们清晰、耐心和详尽的风格我非常喜欢，我建议大家有时间可以读一下(现在读当然更好)。其中的有一部分已经打包在 Perl 的发行版中；特别的如：perldsc (数据结构集锦) 是讲述构建和操纵复杂数据结构的指导教程。

在我们开始讨论例子以前，让我们先来研究一下C或C++中是如何创建结构的。

## 用户定义数据结构

C语言中的struct声明提供了一种用户定义数据类型的记号，然后使用typedef语句为它提供一个新数据类型的别名。Java和C++中使用声明class从基础数据类型中创建新的数据类型。这些结构可以使你用一个统一的名字来表示一组有名属性，而同时又提供对其中的单一属性的存取。

Perl中不存在这种内建的模板功能（注1）。一种常用的方式就是使用散列表来仿真结构类型，如图2-1所示。

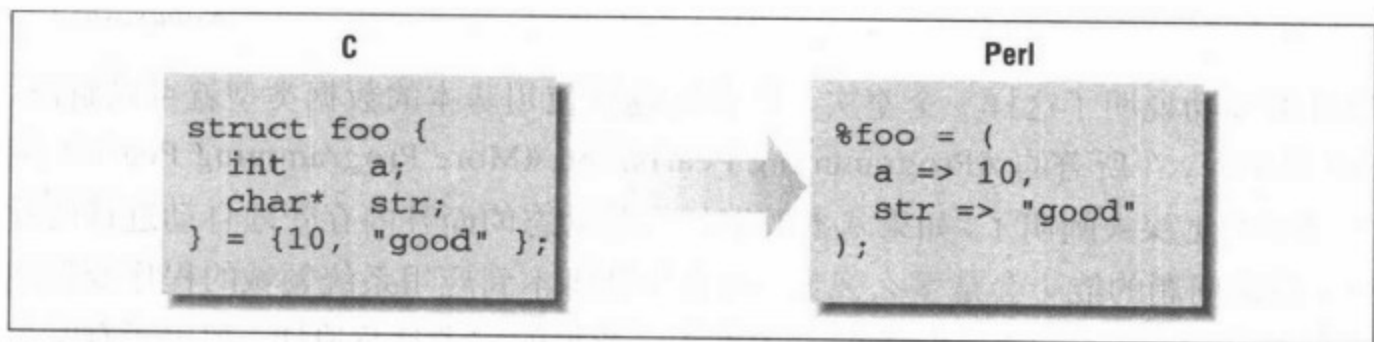


图2-1 用散列表来仿真C语言中的结构类型

Perl中散列表的实现无论在速度还是空间上都相当高效。因为散列键值为常量字符串，Perl只在系统范围内保存一份散列键值的拷贝。这就避免了100个foo结构就要花费100个a和str拷贝的情况。

另一种创建用户定义的属性集合的方法就是使用数组@foo，它更有效率，但却有点笨重：

\$a=0; \$str=1;	# 索引
\$foo[\$a]=10;	# 在C中等价于 foo.a=10
\$foo[\$str]="hello";	# 在C中等价于 foo.str="hello"

请记住，如果一种数据结构用C语言表示要比在Perl中更为简单，而且操作复杂。

注1：我们将在第七章“面向对象编程”中讨论一个提供这项功能的称做ObjectTemplate的模块。

你就可以考虑在C/C++中实现它，而不要在Perl中模仿它的功能。你需要提供一套用以操作数据的C过程。一个简单的名为SWIG的工具（将在第十八章“扩展Perl:第一课”中讨论）就可以轻松的完成此类工作。

你也可以使用pack或sprintf将一系列的值编码成一个复合实体，但是在存取其中单个元素时既不方便，效率也不高(从运行时间上来讲)。如果你是为了节省空间，pack就是一种很好的选择，因为它将一系列值转换为一个标量变量，而无须改变单个元素的内部机器表达形式；sprintf在这方面就不那么高效，因为它把一切都转化为可以打印的表达形式。

## 例子：矩阵

在我们讨论这个例子之前，你必须知道如果你需要的是一个好的，高效的矩阵实现的话，就应该查看一下CPAN中的PDL（Perl数据语言）模块。

为了更好的理解不同的矩阵表达方式，我们将编写用于从数据文件中创建相应数据结构并计算矩阵乘积的例程。数据文件的格式如下所示：

```
MAT1
12      4
10      30      0

MAT2
56
110
```

每个矩阵都有一个标识号和一些数据。我们将使用这些标识号来创建相应的全局变量（@MAT1和@MAT2）。

在Perl中使用数组的数组是最为直观的一种矩阵表达方式，因为Perl不直接支持二维数组：

```
@matrix=(
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
```

```
);
# 将第 1 行, 第 2 列的元素值改为 100
$matrix[1][2]=100;
```

注意这里 `@matrix` 是一个简单的数组, 它的元素为指向匿名数组的引用。也就是说, `$matrix[1][2]` 是 `$matrix[1]->[2]` 的简化表达形式。

例 2-1 中描述了从文件中读取数据并创建数组的数组结构; 它利用符号引用来创建变量 (`@{$matrix_name}`), 并在每次迭代中增加一个对行的引用。我们确信每次迭代时新分配的行是完整的, 虽然 `@row` 局部于所在的代码块, 每次 `if` 语句完成时, 它却能保留下来, 因为我们已经获取了对数组值的引用。(回想一下, 引用记数是对值, 而不是对名字来说的。)

例 2-1: 从文件中读取矩阵数据

```
sub matrix_read_file{
    my($filename)=@_;
    open(F, $filename)|| die "Could not open $filename: $!";
    while($line=<F>){
        chomp($line);
        next if $line=~/^s*$/;           # 跳过空行
        if($line=~/^([A-Za-z]\w*)/){
            $matrix_name=$1;
        }else{
            my(@row)=split(/\s+/, $line);
            push(@{$matrix_name}, \@row); # 将行数组插入到
                                           # 外层的矩阵数组中
        }
    }
    close(F);
}
```

现在让我们使用这种“数组的数组”的结构来计算两个矩阵的乘积。如果你已经忘记了矩阵的乘积是怎么一回事, 那么矩阵  $A_{mn}$  ( $m$  行,  $n$  列) 同矩阵  $B_{np}$  的乘积定义如下:

$$X_{ij} = \sum_{k=1}^n A_{ik} \cdot B_{kj} \quad \text{这里, } i=1 \dots m, j=1 \dots p$$

构成矩阵之积的元素 ( $i, j$ ) 为矩阵  $A$  第  $i$  行与矩阵  $B$  第  $j$  列的对应元素的乘积之和。例 2-2 就是矩阵相乘的 Perl 代码实现。

例 2-2: 矩阵相乘

```

sub matrix_multiply{
    my ($r_mat1, $r_mat2)=@_;          # 获取矩阵的引用
    my ($r_product);                  # 以引用方式返回的矩阵乘积
    my ($r1, $c1)=matrix_count_rows_cols($r_mat1);
    my ($r2, $c2)=matrix_count_rows_cols($r_mat2);
    die "Matrix 1 has $c1 columns and matrix 2 has $r2 rows."
        unless ($c1==$c2);
    for ($i=0; $i<$r1; $i++){
        for ($j=0; $j<$c2; $j++){
            $sum=0;
            for ($k=0; $k<$c1; $k++){
                $sum += $r_mat1->[$i][$k] * $r_mat2->[$k][$j];
            }
            $r_product->[$i][$j]=$sum;
        }
    }
    $r_product;
}

sub matrix_count_rows_cols {          # 返回矩阵的行数和列数
    my ($r_mat)=@_;
    my $num_rows =@$r_mat;
    my $num_cols =@{$r_mat->[0]};    # 假定每行的列数相同
    ($num_rows, $num_cols);
}

```

`matrix_multiply`接受两个矩阵的引用。用`$r_mat->[$i][$j]`来存取单个元素，用`$r_mat->[0]`来存取一行数据。

## 散列表的散列表

如果矩阵大而稀疏（只有一小部分的元素值不为零）的话，那么使用散列表的散列表或许是一种空间利用率更高的表达方式。例如矩阵：

$$\begin{bmatrix} 0 & 0 & 100 \\ 200 & 0 & 0 \\ 0 & 300 & 0 \end{bmatrix}$$

可以用如下方法建立：

```
$matrix{0}{2}=100;
$matrix{1}{0}=200;
$matrix{2}{1}=300;
```

上述代码创建了名为`%matrix`的散列表，它将行号映射到一个嵌套的散列表。只有包含非零元素的行才需要表示出来。而每个嵌套的散列表又将列号映射到所在行与列的实际元素值。同样，也只有在列中存在非零元素时才如此表示。当然，我们必须另外保存矩阵的行数与列数。因为同数组表达方式不同，这些并不能推算出来。由于`%matrix`为散列表，因此它们可以存储为`$matrix{rows}`和`$matrix{cols}`。因为散列表的索引为字符串，这种表达方式也只有在矩阵大而稀疏时才更有效。

为了使前面开发的矩阵例程能够适应新的数据表达方式，似乎只需简单的把所有的方括号替换为大括号即可。不错，这确实可行，但是存在一个微妙的问题。我们假定矩阵的第三行元素全部为零（因此`$r_mat->{2}`并不存在）。现在如果你执行如下命令：

```
$element = $r_mat->{2}{3};
```

Perl 将会自动为`$r_mat->{2}`创建散列表条目，而且该条目的散列表引用为空。（嵌套的散列表并没有创建。）于是仅仅一种简单的查看元素的操作就能吞噬存储空间。这有悖于我们当初选择使用“散列表的散列表”表达方式的初衷。为避免此类情况的发生，我们需要使用`exists`来检测某个散列表元素是否存在，如下所示：

```
If ((exists $r_mat->{$row}) && (exists $r_mat->{$row}{$cols})) {
    ....
}
```

## 其他的矩阵表达方式

如果只有矩阵的列是稀疏的，你可以选择一种数组的散列表结构。还存在一种空间上更为经济的矩阵存储方式，那就是使用单个散列表，但是要付出书写更加复杂代码的代价。如果你将矩阵想像成一种栅格结构，将栅格中的每个单元依次标号，这样每个单元都可以被唯一的数字所标识。那么在一个拥有10行5列的矩阵中，元素（8，4）的标识数为38（7\*5+3），于是可以通过`$r_mat->{38}`来进行



存取。实际上，我们要在第十五章“图形用户界面的例子：Tetris”中使用这种模式(尽管使用的原因更多的是出于方便而不是节省空间)。选择什么样的数据结构依赖于矩阵的大小、性能要求和编码的难易程度等诸多因素。

很明显，如果改变了程序的数据结构，必定要求改变依赖于数据结构的代码。为了限制需要修改的代码数量(当数据结构发生变化时)，只让一小部分过程函数了解数据结构总是不失为一个好的思想。比如，你的程序中有诸如`create_matrix()`，`get_element(mat,i,j)`和`set_element(mat,i,j)`的过程函数，其他过程就不再需要了解数据的内部表达方式了。为将来可能发生的变化编写代码，要比为提高运行时效率编码常常更有益处。我们将在第七章中更多的讨论这个问题。

## 教授，学生与课程

这个例子将向你展示如何表示教授，学生和课程数据的分层记录结构，如何将它们联系起来。假定数据文件如下所示：

```
#file: professor.dat
id           :42343                # 雇员号
Name         :E.F.Schumacher
Office Hours :Mon 3-4,Wed 8-9
Courses      :HS201,SS343          # 讲授课程
...

#file: student.dat
id           :52003                # 学号
Name         :Garibaldi
Courses      :H301,H302,M201       # 学习课程
...

#file: courses.dat
id           :HS201
Description  :Small is beautiful
Class Hours  :Mon 2-4,Wed 9-10,Thu 4-5
...
```

每个“id:”行标识一条新记录的开始。

在各种各样的任务中，我们需要查出是否在教授与学生的时间安排上存在冲突。因为我们的重点是数据表示和Perl引用的语法应用，我们将只会看到问题的部分实现。

## 数据表达

正如我们前面所讲的，散列表对于异构的记录来说是一种很好的表达方式。因此学生的数据结构可以如下实现：

```
$student{42343}=(  
    'Name'=>'E.F.Schumacher',  
    'Courses'=>[]);
```

这里的设计选择有几个微妙之处。

我们本可以将“外关键字”（这里使用了数据库用语）替换成诸如“HS201”的对相应课程数据结构的引用。我们没这样做的原因是，那样就会倾向于直接对这些引用间接访问，与学生有关的代码将会了解课程的数据结构。

我们分别单独为学生数据、课程数据和教授数据维护全局散列表。也就是努力把那些大部分不相关的数据分离开来，以至于对系统的部分改动不会影响到所有人。

这里有一段我们以前没有讨论过的数据结构：时间区间。教授与课程都存在一种“繁忙”或“活动”的时期。这如何表达才更好呢？你或许会选择把诸如“Mon 2-3, Tue 4-6”的行作如下表示：

```
$time_range=(  
    'Mon'=> [2,3],  
    'Tue'=> [4,6]  
);
```

如果你还没有猜出来的话，确实存在一种更加简单的表达方式。问题的关键就在于，既然我们关心的是时间上的冲突，那么系统应当迅速的告知我们在一星期中的某个钟点一个教授或某门课程是否出于“活动”状态。考虑到一星期只有 $24 \times 7 = 168$ 小时，整个星期内的安排可以用长度为21个字节（ $168/8$ ）的位图向量

来表示。如果某位被置位，我们就会知道教授在那个钟点在教课。实际上如果只考虑一周有效的工作时间（如，从上午7点到下午7点，从星期一到星期五），我们还可以进一步减少数据的存储空间。这样一来，就可以减少到8个字节（12小时\*5天/8）。如此做的好处在于，整个时间区间序列缩减为一个包含位图向量的标量变量。另一个就是你可以在两个位图间进行逻辑与操作来检测时间冲突。

确定了表达方式以后，让我们来编写读 *professor.dat* 的数据并创建相应的数据结构的代码。

例 2-3：读 *professor.dat* 数据文件并在内存中创建分层记录

`my (%profs);`      `#prof_read_file()` 将使用文件中的数据来初始化该数据结构

```
sub prof_read_file {
    my ($filename)=@_;
    my ($line, $curr_prof);
    open(F, $filename)||die "Could not open $filename";
    while($line=<F>){
        chomp($line);
        next if $line=~/^s$/;      # 跳过空行
        if($line=~/^id.*:\s*(.*)/){
            # 使用匿名散列表来存储教授的信息
            $profs{$1}=$curr_prof={};
        }elseif($line=~/^Office Hours.*:\s*(.*)/){
            # $1 中包含有 'Mon 2-3, Tue 4-6' 之类的字符串
            $curr_prof->{Office Hours}=interval_parse($1);
        }elseif($line=~/^Courses.*:\s*(.*)/){
            # $1 中包含类似 'HS201, MA101' 之类的信息
            my (@course_taught)=split(/\s,/, $1);
            $curr_prof->{Courses}=\@course_taught;
        }
    }
}
```

注意 `course_taught` 数组局部于代码块。当代码块结束时，`$curr_prof->{Courses}` 依旧保持对该数组的引用。你可以省略一步这么写：

```
$currProf->{Courses}= [split(/\s,/, $1)];
```

我偏爱更具可读性的前者。

interval\_parse 方法解析诸如 “Mon 3-5, Wed 2-6” 之类的字符串并将其转换为前面提到的位串。代码如下：

```
# 一星期中的每个小时（每天从上午7时到下午7时）均在8字节的字符串中占据唯一的一位
# Mon 7-8 占第0位, Mon 6-7pm 占第11位, ..., Fri 6-7pm 占第60位
my %base_hours=(
    mon=>0, tue=>12, wed=>24, thu=>36, fri=>48
);
sub interval_parse {
    my ($interval_sequence)=@_; # 包含有 “Mon 3-5, Tue 2-6”
    my ($time_range)= "";
    foreach $day_hours (split /,/, $interval_sequence){
        # $day_hours 包含 “Mon 3-5” 等等
        my ($day, $from, $to)=
            ($day_hours=~/( [A-Za-z] +.*(\d+)-(\d+)/));
        # 如果 $from 或 $to 小于7, 肯定是下午,
        # 加上12以规范化。然后通过减7将它减少为0 (即7~19变成0~12)。最后,
        # 通过加上每天的“基时”, 把每天的每个小时规范化为每星期的小时
        $to = 19 if $to == 7;
        $from += 12 if $from < 7 ; $to += 12 if $to <= 7;
        my $base = $base_hours{lc $day};
        $from += $base - 7; $to += $base - 7;
        # 此时 Tue 7a.m ==> 12 而 Tue 4 p.m ==> 21
        for ($i = $from; $i < $to; $i++) {
            # 设置相应位
            vec($time_range, $i, 1) = 1;
        }
    }
    $time_range;
}
```

为了检查教授时间安排上的约束, 我们必须计算出教授的工作时间同他或她所教的每门课程之间, 还有课程本身之间的时间重叠, 如例 2-4 中所示。

例 2-4: 检查教授的时间约束

```
sub prof_check_constraints {
    my ($prof) = @_;
    my $r_prof = $profs{$prof}; # %profs created by prof_read_file
    my $office_hours = $r_prof->{Office Hours};
    my $rl_courses = $r_prof->{Courses};
    for $i (0 .. $#{$rl_courses}) {
        $course_hours = course_get_hours($rl_courses->[$i]);
```

```

    if (interval_conflicts($office_hours, $course_hours)) {
        print "Prof. ", $r_prof->{name},
            "Office hours conflict with course $course_taught\n";
    }
    for $j ($i .. $#{$r_courses}) {
        my ($other_course_hours) = course_get_hours($r_courses->[$j]);
        if (interval_conflicts ($course_hours, $other_course_hours)) {
            print "Prof. ", $r_prof->{name},
                ": Course conflict: ", $r_courses->[$i], " "
                    $r_courses->[$j], "\n";
        }
    }
}
}

```

子例程 `interval_conflicts` 简单的比较两个位图，如下所示：

```

sub interval_conflicts {
    my ($t1, $t2) = @_;
    my ($combined) = $t1 & $t2;
    # $combined will have at least one bit set if there's a conflict
    my $offset = length($combined) * 8;
    # start counting down from last bit, and see if any is set
    while (--$offset >= 0) {
        return 1 if vec($combined, $offset, 1);
    }
    return 0;
}

```

值得注意的是，所有内部时间间隔表示的细节均封装在前缀为 `interval_` 的函数中。这些函数因此也就封装了称之为“间隔”的抽象数据类型。在以后的章节中研究模块与对象时，我们要学习到多种将此类代码组织到可重用的实体中的技术。

## 颁奖

举个例子，我们有一份包含以年度和类别划分的学院奖（奥斯卡奖）得主的文本文件，格式如下所示：

```

1995:Actor:Nicholas Cage
1995:Picture:Braveheart

```

```

1995:Supporting Actor:Kevin Spacey
1994:Actor:Tom Hanks
1994:Picture:Forrest Gump
1928:Picture:WINGS

```

我们要提供下列服务：

- 给出年度或类别，打印相应条目（注 2）
- 给出年度，打印出该年度的所有条目
- 给出类别，打印出该类别中的相关年度和所有条目
- 打印出按照类别或年份排序的所有条目

## 数据表示

因为我们要以类别或年度获取相应条目，我们将使用如图2-2所示的双索引结构。

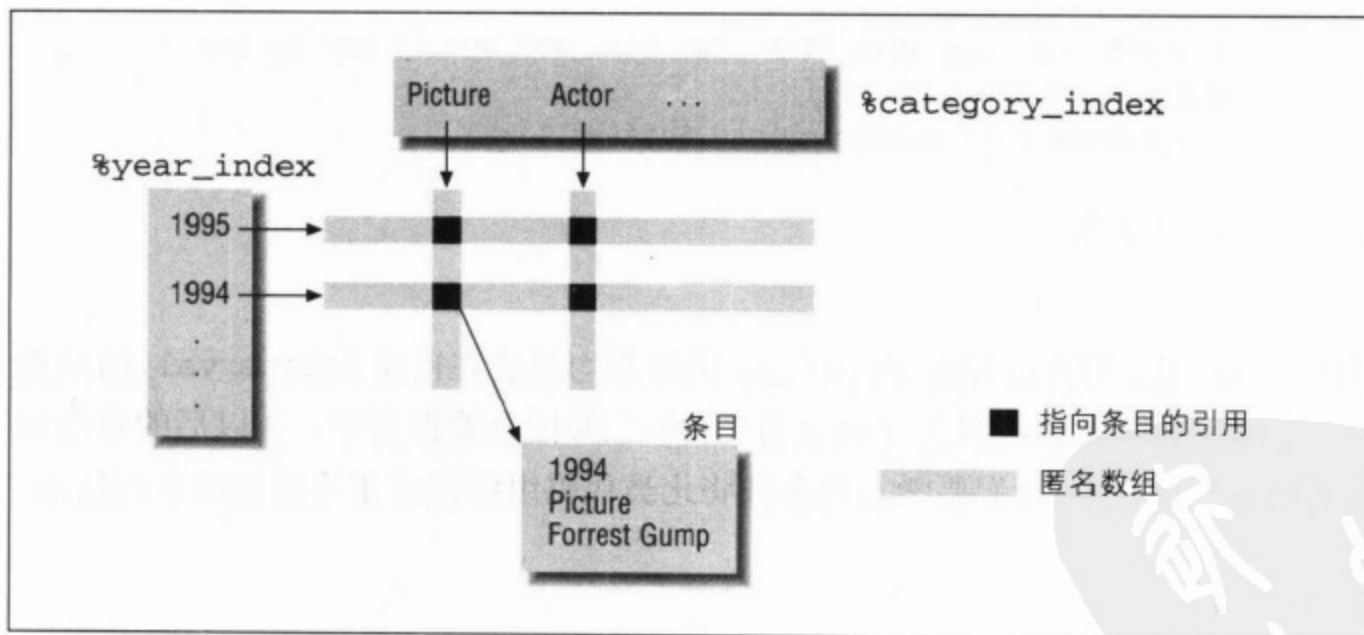


图 2-2 表示奥斯卡奖得主的数据结构

每个条目包含类别、年度和相应的得主名单等信息。我们选择以匿名数组的方式进行存储（匿名散列表也行）。两个索引 %year\_index 和 %category\_index，将

注 2： 要想查看真正的奥斯卡历史数据库，请访问 <http://oscars.guide.com/>（它使用了 Informix 的面向对象数据库 Illustra 来完成这项蹩脚的活儿）。

年度或类别映射到包含有指向具体条目的匿名引用的匿名数组。下面是一种构造该数据结构的实现：

```
open(F, "oscar.txt") || die "Could not open database: $!";
$category_index=(); %year_index=();
while ($line=<F>){
    chomp $line;
    ($year, $category, $name)=split(/:/, $line);
    create_entry($year, $category, $name) if $name;
}

sub create_entry{
    #create_entry (year, category, name)
    my ($year, $category, $name)=@_;
    # 为每个条目创建一个匿名数组
    $rlEntry=[$year, $category, $name];
    # 并将它分别加入两个索引中
    push(@{$year_index{$year}}, $rlEntry);      # 年度索引
    push(@{$category_index{$category}}, $rlEntry); # 类别索引
}
```

注意这里每个 push 语句都做了不少工作。它在索引中创建一个散列条目（如果需要的话），并为此条目预留一个匿名数组（如果需要的话），然后把指向最终数据条目的引用加入数组中。

另一个重要的环节就是，如何使用大括号来保证表达式@{\$year\_index{\$year}}中正确的优先级。如果我们省去那个大括号，那么根据我们在第一章“数据引用与匿名存储”中的“优先级”一节所讲述的规则，首先要计算的将是表达式@\$year\_index而后作为散列表的索引。

## 打印出给定年度的所有条目

我们只需简单的遍历散列表 %year\_index:

```
sub print_entries_for_year{
    my($year)=@_;
    print ("Year: $year \n");
    foreach $rlEntry (@{$year_index{$year}}){
        print("\t", $rlEntry->[1], ":"; $rlEntry->[2], "\n");
    }
}
```

## 打印出所有按年度排序的条目

我们已经知道了如何打印出给定年度的条目。那么，只需将所有的年度数据排序，然后再调用 `print_entries_for_year` 打印出来：

```
sub print_all_entries_for_year{
    foreach $year (sort keys %year_index){
        print_entries_for_year($year);
    }
}
```

## 根据年度与类别打印特定的条目

你可以遍历任何一个索引，但是由于每年的类别要比某一有效类中的年度数少得多，我们将选择遍历索引 `%year_index`：

```
sub print_entry{
    my($year, $category)=@_;
    foreach $rlEntry (@{$year_index{$year}}){
        if ($rlEntry->[1] eq $category){
            print "$category ($year), ", $rlEntry->[2], "\n";
            return;
        }
    }
    print "No entry for $category ($year) \n";
}
```

## 格式化打印工具

在构建复杂的数据结构时，如果手头能有一种供调试用的格式化打印工具就好了。至少有两种选择可用语数据结构的格式化打印。第一种就是Perl调试器本身。它使用了一个在文件 `dumpvar.pl`（它可以在标准库目录中找到）中定义的函数 `dumpValue` 来完成此类工作。我们自己可以来调用它，但是要知道该函数并未公开而且将来可能发生改变。举例说明，我们要格式化输出下面的数据结构：

```
@sample=(11.233, {3=>4, "hello"=>[6,7]});
```



我们可以这么来写：

```
require 'dumpvar.pl';
dumpValue(\@sample);           # 必须以引用方式传递
```

其输出结果如下：

```
0  11.233
1  HASH(0xb75dc0)
   3=>4
   'hello'=>ARRAY(0xc70858)
0   6
1   7
```

我们将在第六章“模块”中讲述 `require` 语句。现在你只需把它当作特别的 `#include`（它不重复加载文件）。

来自 CPAN 的 `Data::Dumper` 模块是另外一种可行的格式化打印方式。在第十章“持续性”中将更详细的讲到这个模块。因此我们在这里先不多讲。这两个模块均能够检测出循环引用，而且可以处理对子程序和 `glob` 的引用。

编写我们自己的格式化打印工具既有趣又很有意义。例 2-5 描述了一种简单的实现，它能够处理循环引用，但不能进一步处理 `glob` 和子程序引用。这个工具的使用如下所示：

```
pretty_print(@sample);          # 不要求是一个引用
```

它会打印出如下结果：

```
11.233
{#  HASH(0xb78b00)
:3 => 4
:hello =>
::    [      # ARRAY(0xc70858)
::    :      6
::    :      7
::    ]
}
```



下面的代码中，包含用来打印特定数据类型的专用过程函数（`print_array`，`print_hash`，`print_scalar`）。用来格式化打印引用的`print_ref`过程，只是根据它要打印的引用类型，将控制交给前面的过程进行打印。而这些过程又可能递归的调用`print_ref`，来处理它们打印过程中的一个或多个引用。

每次遇到一个引用，都要通过散列表`%already_seen`来检查它是否曾被打印过。这样就防止了在碰到循环引用时使过程进入一种无限循环中。所有函数均通过操纵全局变量`$level`和调用过程`print_indented`，来恰当的处理格式打印的缩进，并打印出给定的字符串。

#### 例 2-5: 格式化打印工具

```
$level = -1; # Level of indentation

sub pretty_print {
    my $var;
    foreach $var (@_) {
        if (ref ($var)) {
            print_ref($var);
        } else {
            print_scalar($var);
        }
    }
}

sub print_scalar {
    ++$level;
    my $var = shift;
    print_indented ($var);
    --$level;
}

sub print_ref {
    my $r = shift;
    if (exists ($already_seen{$r})) {
        print_indented ("{$r (Seen earlier)}");
        return;
    } else {
        $already_seen{$r}=1;
    }
    my $ref_type = ref($r);
    if ($ref_type eq "ARRAY") {
```

```
        print_array($r);
    } elsif ($ref_type eq "SCALAR") {
        print "Ref -> $r";
        print_scalar($$r);
    } elsif ($ref_type eq "HASH") {
        print_hash($r);
    } elsif ($ref_type eq "REF") {
        ++$level;
        print_indented("Ref -> ($r)");
        print_ref($$r);
        --$level;
    } else {
        print_indented ("$ref_type (not supported)");
    }
}
```

```
sub print_array {
    my ($r_array) = @_;
    ++$level;
    print_indented ("[ # $r_array");
    foreach $var (@$r_array) {
        if (ref ($var)) {
            print_ref($var);
        } else {
            print_scalar($var);
        }
    }
    print_indented ("]");
    --$level;
}
```

```
sub print_hash {
    my($r_hash) = @_;
    my($key, $val);
    ++$level;
    print_indented ("( # $r_hash");
    while (($key, $val) = each %$r_hash) {
        $val = ($val ? $val : '');
        ++$level;
        if (ref ($val)) {
            print_indented ("$key => ");
            print_ref($val);
        } else {
            print_indented ("$key => $val");
        }
    }
}
```



```
        }
        --$level;
    }
    print_indented ("}");
    --$level;
}

sub print_indented {
    $spaces = ": " x $level;
    print "${spaces}${_[0]}\n";
}
```

`print_ref`只是简单的打印出它的参数(一个引用),或者在发现它以前曾经见过这个引用时返回。如果你阅读这些代码的打印输出时,你会发现很难确定某个引用指向的是哪一个数据结构。作为练习,你可以试着去编写一个更好的格式化打印工具,使它能够用数字来标识相应的结构。如下所示:

```
:hello =>
::      [      # 10
::      :      6
::      :      7
::      ]
:foobar => array-ref      # 10
}
```

这里的数字10是自动产生的标识符,它要比诸如`ARRAY(0xc70858)`的这种形式更容易辨认。

## 相关资源

FMTYEWTK 系列指导教程 (Far More Than You Ever Wanted To Know) .Tom Christiansen。

可从下面的地址获得: <http://language.perl.com/info/documentation.html>。

本章简介:

- Perl 变量, 符号表和作用域
- Typeglob
- Typeglob 与引用
- 文件句柄, 目录句柄与打印格式

## 第三章

# Typeglob 和符号表

我们就是些符号, 而且生活在符号中。

——爱默生 (译注 1)

本章将讨论 typeglob、符号表、文件句柄、打印格式以及动态作用域与词法作用域的差异。乍一看来, 这些问题似乎毫不相关, 但是实际上, 它们同 typeglob 与符号表关系密切。

Typeglob 非常有用。有了它们, 我们就可以创建符号的别名, 这是一个广泛应用在大量自由模块中的 Exporter 模块的基础。Typeglob 还可以用作普通引用的别名, 这样你就不必再进行“间接访问”了。这不光使程序更容易阅读, 而且速度也更快。同时, 如果你并不理解 typeglob 是如何工作的就去使用它, 则会导致一种让人头痛的称作“变量自杀”的问题。这也是为什么大多数 Perl 文献很少谈及 typeglob 的原因。

与 typeglob 和符号表密切相关的是动态与词法作用域的差异 (前者使用 local 而后者使用 my)。这些差异又引出了几个有用的专业用语。

本章是唯一开篇不向你展示可以直接使用的实例, 而直接讲述内部工作的细节的一章。希望这样能够让你更容易理解随后的讨论。

---

译注 1: 爱默生 (1803 - 1882), 美国思想家、散文作家、诗人。

## Perl 变量，符号表和作用域

变量要么就是全局性的，要么就是词法的 (*lexical*，指那些以 `my` 语句限定的变量)。本节我们要讨论这两种情况在内部是如何表示的。我们先来讨论全局变量。

Perl 有一个在其他语言中一般见不到的奇怪特性：你可以使用同样的名字来标识数据或非数据类型。例如，标量变量 `$spud`，数组 `@spud`，散列表 `%spud`，子例程 `&spud`，文件句柄 `spud` 和打印格式名 `spud` 可以同时有效而且完全相互独立。换句话说，Perl 为每一种类型的实体提供不同的名字空间。我无法解释该特性存在的意义。实际上，我觉得这是一种不可靠的机制，而且建议你为程序中的每个逻辑实体起一个不同的名字。不然，你肯定对不起那个将来进行代码维护的可怜家伙（也许就是你自己来维护它）。

Perl 使用符号表来把标识符名（就是去掉前缀的“`spud`”字符串）（注 1）映射到相应的值。但是散列表不允许重复键值，因此你不可能使散列表中拥有相同名字的两个条目指向两个不同的值。于是，Perl 就在符号表与其他数据类型之间，放置了一个叫做 `typeglob` 的数据结构，如图 3-1 所示。我们可以使用同一个名字来访问一组指向值的指针，其中每个指针对应一种值类型。在一种只有一个唯一标识符名字的典型情况下，所有这些指针只有一个非空。

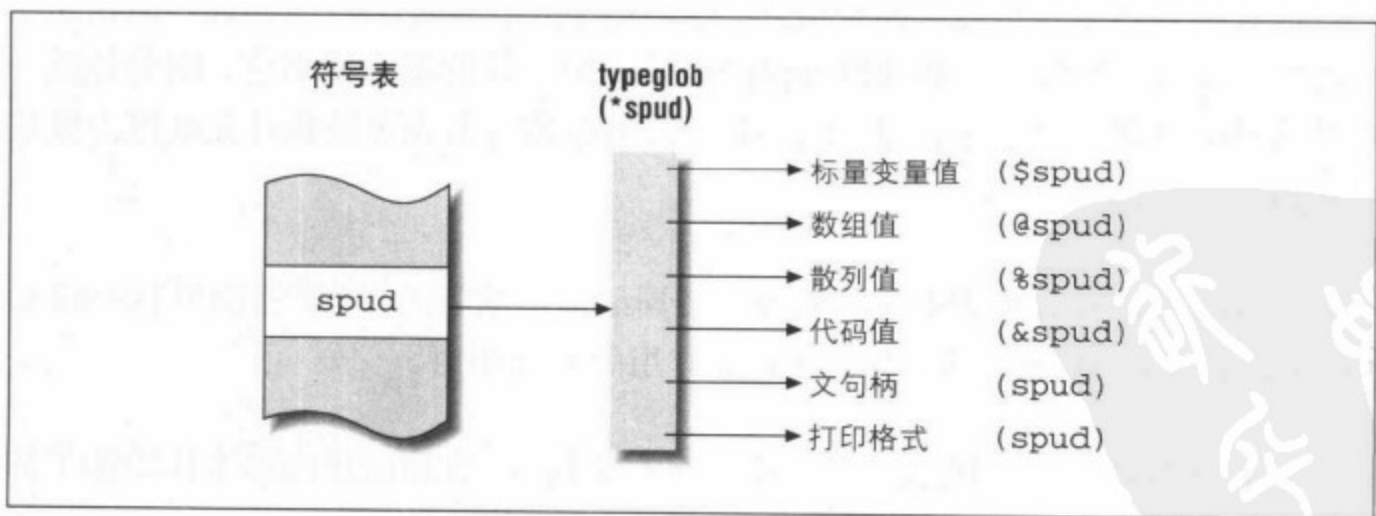


图 3-1 符号表与 typeglob

注 1：实际上是每个包一个符号表，每个包的名字空间是不同的。暂时先不要理会这种差异。我们将在第六章“模块”中重新讨论这个话题。

Typeglob 是一种可以在脚本程序中使用的真实的数据类型，它拥有前缀“\*”，尽管你可以把它想像成一种代表所有值的通配符，这些值拥有相同的标识符，但是实际上并没有执行任何匹配操作。你可以进行 typeglob 赋值，把它们保存在数组中，为它们创建局部化版本或者将它们打印出来，你可以像对待基础数据类型那样使用它。我们一会儿再进行详细的讨论。

## 词法变量

词法变量（那些以 `my` 语句限定的变量）并不在符号表中出现。每个代码块和子例程都有一个或一串（通常情况下只有一个，但是如果子例程递归的话就会多一些）变量数组，它们被称为“便签簿（scratchpad）”。每个词法变量都在“便签簿”中占据一项。实际上，拥有相同名字的不同类型变量，如 `$spud` 和 `%spud`，占据不同的表项。由于子例程的词法变量独立于其他过程，因此它们是真正的局部变量。我们将在第二十章“Perl 的内部工作”中进一步讨论这个问题。

## 词法与动态作用域的比较

有两种方式将私有数值传入子例程或代码块。第一种是使用 `local` 操作符，但它只限于全局变量；它先将变量值保存，然后在代码块结束时再恢复原有值。另外一种方式就是 `my`，它不但又新建了一个变量，而且还把它置为只对代码块可见。

表面上看来，`local` 和 `my` 的效果相同，如：

```
$a=20;                                # 全局变量
{
    local ($a);                        # 先保存 $a 原先的值，新值为 undef
    my (@b);                           # Lexical 变量
    $a=10;                             # 改变 $a 的新值
    @b=("wallace", "grommit");
    print $a;                          # 打印结果为 "10"
    print "@b";                        # 打印结果为 "wallace grommit"
}
# 代码块结束。又回到全局域中，此时只有 $a 有效
print $a;                             # 打印出 "20"，为原先的值
print @b;                             # 打印警告信息，因为不存在全局变量 @b
```

由于 `local` 语句使得该全局变量在代码块终结时被释放并以原来的值重新创建。

虽然它们的用法相同，但是 `local` 与 `my` 之间有一个很重要的差别。`my` 声明将创建真正的局部变量，这有点像 C 语言中的自动变量。它被称作词法联编。变量只在声明它的代码块中可见（代码块从词法上定义了这种界限）。它对代码块中调用的其他子例程来说是无效的。

与 `my` 不同，`local` 操作符并不创建新的变量。当你把它作用于全局变量时，它将会先把全局变量值隐藏起来，并在代码块终结时予以恢复。因为该变量本身就是全局变量，因此它的值不仅对使用 `local` 的代码块有效，而且对于代码块中调用的过程同样有效。考虑下面的例子：

```
$x = 10;
first();

sub first{
    local ($x) = "zen";           # $x 仍然是全局变量，而且现在有了新值
    second();
}
sub second(){
    print $x;                     # 将打印出全局变量的当前值，"zen"
}
```

在全局域中，我们首先调用 `first`，它将全局变量局部化并赋予一个新值（字符串“zen”），然后调用 `second`，`second` 看到的 `$x` 是由 `first` 设置的当前值。这个过程被称做作用域动态化（dynamic scoping），因为由 `second` 所看到的当前值取决于特定的调用栈。该特性在实际运用中有时很令人费解。比如如果你编写了另一个其中含有 `local $x` 的子例程并在其中调用 `second` 子例程，那么 `second` 将使用当前这个子例程中的 `$x` 版本。

换句话说就是 `local` 以临时的方式保存当前的新值，它并没有改变变量的本质（变量仍然是全局变量）。而 `my` 则创建新的局部变量。因此你可以这么写：

```
local $x{foo};           # 先将 $x{foo} 的值暂存起来
```

而不能这么写：



```
my $x{foo}          # 错。$x{foo}不是一个变量
```

我们建议你尽可能的多使用`my`，因为你通常需要的是词法作用域。同时，我们将在第二十章了解到词法作用域的变量要比动态作用域的变量速度快。

### 到底什么时候需要使用 `local`?

`Local`保存变量的值并在代码块结束时恢复值，这一事实引出了一个很有用的专业用语：局部化内建变量。下面是一个将用于保存程序命令行参数的内建变量`@ARGV`局部化的例子：

```
{# 标识代码块的开始
  local(@ARGV) = ("home/alone", "/vassily/kaninski");
  while(<>){
    # 对每个文件进行迭代处理
    print;      # 使用print 进行处理
  }
} # 代码块的终结。此时 @ARGV 原来的值将被恢复
```

钻石操作符（`<>`）需要与全局变量`@ARGV`配合工作，因此它在符号表中通过`typeglob`查找条目`ARGV`，可是它并不知道`@ARGV`的值已经被`local`换成了一个新的数组值（注2）。钻石操作符把数组中的每个元素当作文件名，打开文件，在每次迭代中从中读取一行然后在需要时继续处理下一个文件。在该代码块结束时，恢复`@ARGV`原来的数值。本例中不能使用`my`操作符，因为它将创建一个全新的变量。

这个技巧对于其他内建变量同样有效。比如变量`$/`中保存输入记录分隔符（默认为“`\n`”）。钻石操作符使用它来返回下一条记录（默认为下一行）。如果你取消该变量值的定义（`undef`），整个文件就会被一口气全部读进来。为了免于保存与恢复`$/`的原始值，你可以像下面这样使用`local`操作符：

```
{
  local $/=undef;      # 保存 $/ 的原始值并将它设置为 undef
  $a=<STDIN>;          # 将标准输入的内容全部读入 $a 中
}
```

注2：出于效率原因，Perl在运行时并不进行符号表查找。编译阶段会确保相应的操作码知道要访问哪个`typeglob`。第二十章有更多这方面问题的讨论。

local 也可以用来局部化 typeglob，这也是获取局部化文件句柄、打印格式和目录句柄的唯一一种方式。

## Typeglob

我们前面提到 typeglob 可以被局部化（只能使用 local），还可以相互赋值。通过 typeglob 赋值可以创建标识符的别名。例如：

```
$spud = "Wow!";
@spud = ("idaho", "russet");
*potato = *spud;           # 使用 typeglob 赋值创建 spud 的别名 potato
print "$potato \n";        # 将打印出 "Wow!"
print @potato, "\n"        # 将打印出 "idaho russet"
```

一旦进行了 typeglob 赋值，所有名字为 spud 的实体均可以使用 potato 来指代——这两个名字可以自由互换。也就是说，\$spud 与 \$potato 表示同样的东西，而且子例程 &spud 与 &potato 也是如此。图 3-2 描述了 typeglob 赋值后的情形；符号表中的那两个条目均最终指向相同的 typeglob 值（注 3）。

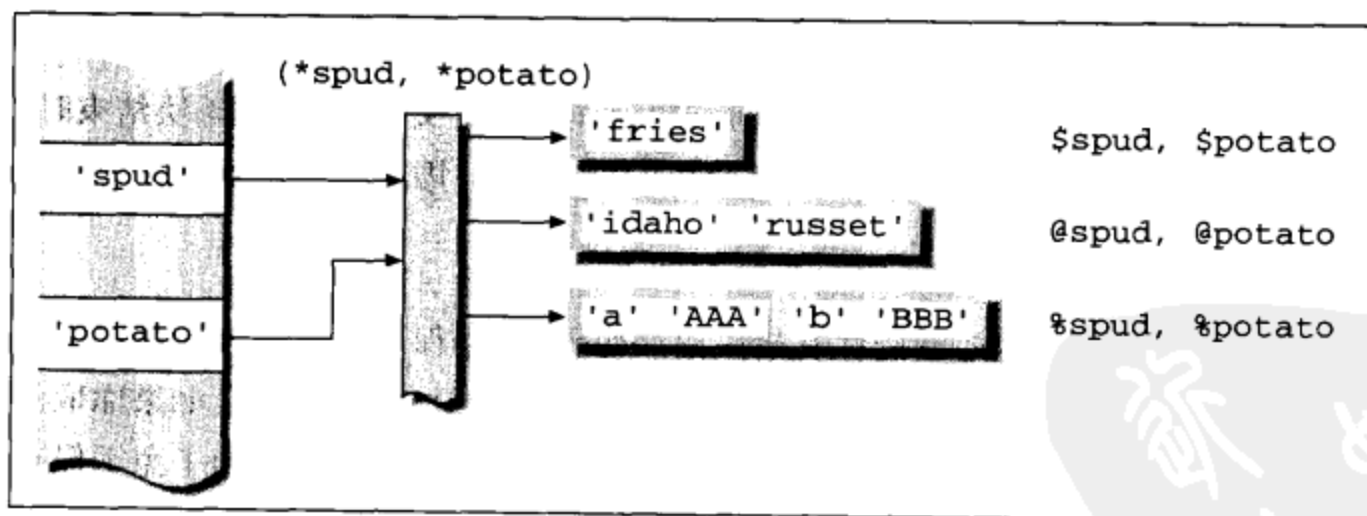


图 3-2 将 \*spud 赋值给 \*potato，它们的符号表条目均指向同一个 typeglob

这种别名讲一直持续到该 typeglob 被重新赋值或删除。（我们马上就要学到如何删除一个 typeglob。）在上面的例子中虽然没有名为 spud 的子例程，但是如果我

注 3： 这里稍微有些简化，我们将在第二十章澄清这一点。

们在 typeglob 赋值后再定义它，我们仍然可以通过 potato 来访问到它。其实这种别名反过来同样可行。如果你将一个新的表赋值给 @potato，同样可以自动的通过 @spud 来存取。

## 临时别名

目前暂时还没有一种简单而直观的方法，用于删除由 typeglob 赋值所产生的别名（你当然可以将它重新赋值）。然而你可以通过 local 创建临时的别名，因为它将在代码块结束时，恢复 typeglob 的原始值。例如：

```
$b = 10;
{
    local *b;           # 保存 *b 的值
    *b = *a;            # 创建 a 的别名 b
    $b = 20;            # 这等价于修改 $a 的值
}
print $a;              # 打印出 "20"
print $b;              # 打印出 "10"
```

local \*b 使得对所有名为“b”的变量的修改局部化；也就是说，它首先将 \*b 的所有值指针保存起来，然后将它们统一替换为值 undef。这种状况一直持续到该代码块结束，而后恢复所有名为“b”的变量的值（\$b 的值又成为 10）。由于别名的存在（\*b = \*a），赋值操作 \$b=20 将会同时改变 \$a 与 \$b 的值。于是当代码块结束时，包含新赋值的 \$a 就被保留下来。

在讨论这个问题时，有一点很重要，那就是词法变量与符号表间没有任何关系；因此使用 my 来局部化 typeglob 将会导致编译错误：

```
my(*F);
```

该脚本程序将会打印出错误信息“Can't declare ref-to-glob cast in my（无法用 my 声明 ref-to-glob 转换）”后退出。

## 使用 typeglob 别名

在这一节我们要讨论几个很好的运用 typeglob 别名的例子。

## 高效的参数传递

因为别名不存在间接访问操作，因此它要比引用的速度快一些。例如：

```
$a = 10;
*b = *a; $b++;      #1. 通过 typeglob 方式间接的使 $a 加一
$r = \ $a; $$r++;   #2. 通过引用方式间接的使 $a 加一
```

在我的 PC 机上，第一种方式的速度要比第二种方式快一倍半。

下面的例子代码使用 `typeglob`，来高效地以引用方式将数组传递给子例程 `DoubleEachEntry`，而该子例程又将数组的每个元素加倍：

```
@array = (10, 20);
DoubleEachEntry(*array);      #@array 与 @copy 是等价的
Print  "@array \n";          # 打印结果为 20 40
Sub DoubleEachEntry{
    #$_[0]中包含有 *array
    local *copy = shift;      # 创建局部化的别名
    foreach $element(@copy){
        $element*=2;
    }
}
```

大家注意到只向子例程传递了一个参数。Typeglob `*copy` 首次出现时即被创建，但是由于它在 `local` 语句前并不存在，因此它在符号表中的相关条目及其本身将在代码块结束时被删除。

碰巧的是，这段代码利用了 `foreach` 语句的内部特性——`foreach` 语句在内部相继为 `@copy` 中的每个元素创建名为 `element` 的别名，因此修改 `$element` 就是在修改 `@copy` 中的元素（同样也就是在修改 `@array`）。

你不能使用词法作用域的数组作为 `DoubleEachEntry` 的参数，因为词法变量并没有与之相关的 `typeglob`。然而这种限制很容易被打破。奇怪的是，`typeglob` 竟会与引用等价。你可以向需要以 `typeglob` 为参数的子例程传递普通的引用，而且这样工作的非常好（我们将在“`typeglob` 与引用”一节更多的谈到这个问题）。所以你可以如下所示，将词法作用域的数组传递给子例程 `DoubleEachEntry`：

```
my @array = (1, 2, 3);
DoubleEachEntry(\@array);      # 这里不是传递 *array, 那样的话则不能工作
```

## 在命令行上应用别名

我经常将Perl解释器嵌入自己的C/C++程序中,以提供一种功能强大的类似于命令行的前端。虽然我喜爱在Perl脚本程序中使用长而具有描述性的子例程名,但是在命令行前端敲入这些东西,确实让人头疼。别名在这里会非常有用:

```
sub a_long_drawn_out_sub_name{
    print "A sub by any other name is still a sub \n";
}
*f= *a_long_drawn_out_sub_name;      # 创建了一个别名
```

现在在命令行上敲入 `f()` 就等价于调用原来的那个子例程,手腕就轻松多了。

## 使用别名来创建友好的预定义变量

我们要反其道而行之。Perl中存在许多晦涩难懂的内建变量,如 `$!`, `$/` 和 `$@`,于是许多人宁愿使用那些长而更具描述性的名字。Perl标准库中的 *English.pm* 模块能够帮助解决这个问题,它提供长而友好的别名,如 `$ERRNO`、`$INPUT_RECORD_SEPARATOR` 和 `$EVAL_ERROR`。举例如下:

```
use English;                      # 引入名为 English.pm 的模块名
# 试图删除一个并不存在的文件
unlink('/tmp/foo');
if ($ERRNO){                       # 这里使用 $! 的别名 $ERRNO
    print "$ERRNO", "\n";         # 将打印出 "No such file or directory"
}
```

(我们将在第六章“模块”中讲述包及 `use` 语句的使用。)我觉得从一开始就应该保留这些众所周知的名字,而不是去记那些希奇古怪的变量名,再配一套助记清单。有人会争辩说,你可以使用同样的方式来为其他语言创建别名(如:“`use Dutch;`”),但是不管怎么说,其他系统调用是用英文写的,我不认为仅仅为你要记忆东西的一小部分提供特定的别名有什么意义。

## 别名存在的问题:变量自杀

local 并不真正创建新的变量(它只是暂时的为全局变量换上一个新值),当别名在这种情况中使用时,常会使一些变量产生莫名其妙的值,而这些变量似乎从来碰都没碰过。考虑下面的简单例子:

```
$x=10;
foo(*x);
sub foo {
    local(*y) = @_;
    print "Before value of y: $y \n";
    local($x)=100;
    print "After value of y: $y \n"}
```

这段程序将打印出如下结果:

```
Before value of y : 10
After value of y : 100
```

你能理解其中的奥妙吗?很明显, \$y 在这两条 print 语句间从未改动过,但是它的值却似乎被改变了。提示:它反映的是 \$x 的值。

让我们跟踪一下都发生了什么:

\$x = 10;	# 为全局变量 \$x 赋值
	# 现在函数被调用了
local *y = *x;	# 保存全局 *y 的所有值,并将它设置为 *x 的别名
print "before value"	# 由于别名的存在, \$y 现在等价于 \$x
	# 因此这条语句的打印结果为 10
local \$x = 100	# 这点非常重要:local 保存了 \$x 的值(10)
	# 并替换为新值 100。要注意,它并没有创建新的 \$x 变量
print "after value"	# 但是由于 *y 依旧是 *x 的别名,因此, \$y 现在的值为 100

下面的例子中别名与 local 语句所导致的问题更加微妙:

```
foreach $f (10, 20, 30){
    foo(*f);
}
sub foo {
    local(*g)=@_;
```

```
    $g++;  
}
```

这段代码将打印出错误信息：“Modification of a read-only value attempted at try.pl line 6 (try.pl 第6行中企图修改只读值)”。

单步解析如下：处于效率的考虑，foreach操作符在每次迭代中，均将\$*f*设置为列表中下一个元素的别名，这里每个元素都是常量。而子例程foo将\**g*设置为\**f*的别名，也就是说\$*g*是一个常量的别名。因此，操作\$*g*++将导致错误。

心得：如果你需要的是真正的局部变量，就使用my操作符。要谨慎的使用typeglob别名和local操作符。

## Typeglob 与引用

你或许已经注意到typeglob和引用均指向值。变量\$*a*可以被简单的视为一个typeglob的间接访问\${\**a*}。因此，Perl使这两种表达式\${\ \$*a*}和\${\**a*}指向同一个标量变量值。Typeglob与普通的引用等价有一些有趣的性质，并引出了三个有用的专业用语。描述如下：

### 别名的选择性

前面，我们已经看到一条诸如\**b*=\**a*的语句，如何使得名为“*a*”的一切可以用“*b*”来指代。使用引用的语法，可以创建有选择的别名：

```
*b = \ $a;           # 将一个标量变量的引用赋值给 typeglob
```

这样Perl只安排\$*b*与\$*a*相互为别名，而@*b*与@*a*（或者&*b*与&*a*等等）则不是。

### 常量

我们可以通过对常量的引用来创建只读变量，如下所示：

```
*PI=\3.1415927;
```

```
# 现在试着修改它
$PI=10;
```

Perl 将打印错误信息:“Modification of a read-only value attempted at try.pl line 3 (try.pl 第 3 行中企图修改只读值)”。

## 为匿名子例程起名

我们将在下一章中讨论匿名子例程,因此你以后可能会再回来查看这个例子。

你如果觉得间接的通过引用来调用子例程(&\$rs())很不舒服的话,那么你可以赋给它一个名字以求方便。

```
sub generate_greeting{
    my($greeting)=@_;
    sub{ print "$greeting world \n";}
}
$rs=generate_greeting("hello");
# 为了不以&$rs()的形式调用子例程,你可以给它起个名字
*greet=$rs;
greet();           # 它等价于调用&$rs()。打印结果为"hello world\n"
```

当然,你也可以为其他类型的引用起名。

## 对 typeglob 的引用

我们已经看到引用与 typeglob 是如何等价的了(从引用可以赋值给 typeglob 这一点上来说)。Perl 还允许你创建的 typeglob 的引用,你只需依照惯例以反斜杠作为前缀即可:

```
$ra=\*a;
```

在实际应用中,对 typeglob 的引用并不常见,这是因为以 typeglob 方式进行传递本身就很高效。这非常类似于普通标量变量的情形,那里不需要传递对标量变量引用来提高效率。



## 文件句柄，目录句柄及打印格式

内建函数 `open` 和 `opendir` 分别用来初始化文件句柄和目录句柄：

```
open(F, "home/calvin");
opendir(D, "/usr");
```

符号 `F` 和 `D` 为用户自定义的标识符，但是没有前缀符号。不幸的是，这些句柄没有其他诸如标量变量、数组和散列表等重要数据类型所具备的一些基本特性——你不能进行句柄赋值，还不能创建局部化的句柄（注 4）：

```
local (G);          # 无效
G=F;                # 同样无效
```

在我们继续讨论之前，要清楚的重要一点就是，标准 Perl 发行版中自带了一个名为 `FileHandle` 的模块，它提供了一种面向对象版本的文件句柄。它允许你创建文件句柄“对象”，相互之间进行赋值和创建局部于代码块的版本。类似的，目录句柄则由 `DirHandle` 模块负责。现在提倡开发人员使用这些新的机制，而不是我们后面要讲到的方法。但是你还是要多费一番力气来学习后面的讨论，因为有大量的自由软件代码中使用了这种结构；实际上，那些标准块如 `FileHandle`，`DirHandle` 和 `Symbol`，还有整个分层 IO 模块都是建立在这种结构基础上的。

为什么能够进行句柄赋值和创建局部化的句柄会这么重要呢？不能赋值，你就不能将文件句柄作为参数传递给子例程，或将它们保存在数据结构中；没有局部化的文件句柄，你就不能创建含有 `open` 操作的递归的子例程（它被用于处理包含文件，包含文件本身还可以包含其他文件）。

一种简单的解决方案就是使用 `typeglob` 赋值。就是说，如果你要急切的表达下面的这种意思：

```
G=F;
# 或者
local(F);
```

注 4： 我不知道为什么文件句柄没有一种标准的前缀符号，以及其他数据类型所享有的一些别的功能。

你则可以以 `typeglob` 的方式如下编写:

```
*G=*F;  
# 或者,  
local(*F);
```

与之类似,如果你要把文件句柄保存在数据结构中或是创建对它们的引用,你可以使用相应的 `typeglob`。所有接受文件句柄的 I/O 操作同样也接受 `typeglob` 引用。让我们看一下我们可以通过文件句柄赋值和创建局部化版本(当然要使用 `typeglob`)来做什么。

## I/O 重定向

下面的例子显示如何进行简单的 I/O 重定向:

```
open(F, '>/tmp/x')||die;  
*STDOUT=*F;  
print "hello world \n";
```

`print` 函数以为它在向标准输出 `STDOUT` 打印信息,然而最终却输出到了一个打开的文件中。因为 `STDOUT` 的相关 `typeglob` 被以别名的方式指向 `F`。如果你希望重定向是暂时的,你就可以创建 `*STDOUT` 的局部化版本。

## 把文件句柄传递给子例程

下面的一段代码将文件句柄传递给一个子例程:

```
open(F, "/tmp/sesame")||die $!;  
read_and_print(*F);  
sub read_and_print{  
    local (*G)=@_;          # 文件句柄 G 与文件句柄 F 相同  
    while(<G>){print;}  
}
```

你或许会感到奇怪,为什么不以同样的方式包装 `open`, 它毕竟也是个子例程而且同样需要以文件句柄为参数。其实对于诸如 `open`, `read`, `write` 和 `readdir` 等内建函数, Perl 会自动传递符号的 `typeglob` (例如,而不是名为 “F” 的字符串)。

## 局部化文件句柄

让我们来看一个遍历 C 头文件中包含声明的子例程，代码如下所示。子例程 ProcessFile 查看文件的每一行，如果发现匹配的 #include 声明，它就会析取文件名然后递归的调用自己。因为它在原先的文件中有更多的行要处理，所以它不能关闭文件句柄 F。如果 F 是全局性的，它就不能够再被用来打开别的文件。因此我们使用 local(\*F) 将它局部化。这样，每次递归调用的 ProcessFile 将会拥有自己唯一的文件句柄值。

```
sub ProcessFile {
    my ($filename)=@_;
    my ($line);
    local(*F);           # 保存原先 typeglob 的值(也就是许多值中的文件句柄)
    open(F, $filename)||return;
    while($line =<F>){
        # 与前面相同
        .....
    }
    close(F);
}
```

尽管我们还没有讨论过包，但是值得看看 FileHandle 模块在这种情况下是如何使用的：

```
use FileHandle;
sub ProcessFile {
    my ($filename)=@_;
    my ($line);
    my $fh =new FileHandle;           # 创建局部句柄
    open($fh, $filename)||return;
    while($line=<$fh>){
        .....
    }
    close($fh);
}
```

## 以字符串为句柄

实际上 typeglob 和 FileHandle 模块的对象并不是唯一的解决方案。所有接受一个句柄的 Perl I/O 函数都可以接受一个字符串。考虑下面的代码：

```
$fh = "foo";  
open ($fh, "< /home/snoopy") ;  
read ($fh, $buf, 1000);
```

当 `open` 检查其参数时，会发现原来应该是一个 `typeglob` 的地方出现了一个字符串。这种情况下，它会自动的用那个名字创建一个 `typeglob`，然后像往常一样继续处理。与之相似，当 `read` 接收的是一个字符串而不是 `typeglob` 时，它会从符号表中查找相应的 `typeglob`，接着是那个内部的文件句柄，随后继续读取相应的文件。这种额外的查找要比使用一个纯粹的符号稍慢一些，但是如果你是以适当大小的数据块来进行的 I/O 操作的话，这点时间则是微不足道的（最佳的数据块尺寸对于不同的系统是不同的）。



#### 本章简介:

- 子例程的引用
- 使用子例程引用
- 闭包
- 使用闭包
- 与其他语言的对比
- 相关资源

## 第四章

# 子例程引用 与闭包

许多人都被邀请了，但是到场的却没几个。(译注1)

——*Sister Mary Tricky*

和普通变量一样，子例程 (subroutine) 可以是有名的也可以是匿名的，而且 Perl 语法上支持对任何一种类型的引用。这种引用很像 C 语言中指向函数的指针。它可以被用来创建如下复杂的结构：

- **调度表 (dispatch table)**。它是一种将事件映射到子例程引用的数据结构。每当事件发生时，就会通过一张调度表来查找与之相关的子例程。这在创建诸如大型高效的交换语句、有限状态机、信号处理程序和图形用户界面工具包时非常有用。
- **高等级子例程 (High-order procedure)**。一个高等级子例程以其他的过程为参数 (有点像 C 的库函数 `qsort`)，或者返回新的过程。后一种情况只出现在诸如 Perl、Python 和 LISP 等解释型语言中 (嘿，LISP 程序员们，你们有 `lambda` 函数)。
- **闭包 (Closure)**。闭包是这样一种子例程，创建时，它将包含其子例程的环境打包 (包括所有它需要的和不局部于其本身的变量)。

译注 1：此句原文为：“Many are called, but few are called back.”，call 和 call back 有调用和回调的双关意。译文无法体现。

## 子例程引用

子例程并没有什么特别或神奇的地方。在这一节我们要学习如何创建对有名的和匿名的子例程的引用，以及如何对它们进行间接访问。

### 对有名子例程的引用

我们前面就已经将过，要创建对现存变量的引用，只需给它加上反斜杠前缀。对子例程也大致如此，如 `\&mysub` 就是对 `&mysub` 的引用。又比如：

```
sub greet {  
    print "hello \n";  
}  
$rs=\&greet;           # 创建对子例程 greet 的引用
```

有重要的一点需要说明，我们并没有在此调用子例程 `greet`，这和创建对标量变量的引用时一样，也没有计算标量变量的值。

与下面带有圆括号的代码作一对比：

```
$rs=\&greet();
```

该表达式的结果可能不是你所希望的。它调用 `greet`，然后创建了一个对其返回值的引用，也就是子例程中最后一个表达式的值的引用。由于 `print` 最后执行并返回 1 或 0（表明是否成功打印了后面的值），因此该表达式的结果就是一个对值为 1 或 0 的标量变量的引用。这种类型的错误偶尔会使你怀念起类型安全的好处。

总结：在创建对子例程的引用时不要使用圆括号。

### 对匿名子例程的引用

你只需省略掉子例程声明中的过程名即可创建匿名子例程。除了这一点外，其声明同有名子例程完全一样。

```
$rs=sub{
```

```
print "hello \n";  
};
```

上面的表达式将返回一个对新声明的子例程的引用。注意,由于它是一个表达式,因此需要最后的分号,这和声明有名子例程不同。

## 对子例程引用的间接访问

对子例程引用的间接访问 (dereference) 将会间接的调用该子例程。和数据引用一样, Perl 并不关心 `$rs` 指向的有名还是无名的子例程; 间接访问操作对两种情况都适合。

通过在 `$rs` 前面加上合适的前缀 “&” 来表示间接访问操作, 并没有什么希奇的:

```
&$rs(10,20);      # 间接的调用子例程
```

就是这么简单。

正如你可以在数组和散列表中使用 “->” 语法一样 (`$rs->[10]` 或 `$rh->{'k2'}`), 你可以像下面这样通过引用来间接地调用子例程:

```
$rs->(10);
```

实际上, 如果中间的调用同样返回对子例程的引用的话, 子例程调用可以链接起来, 例如:

```
$rs=&test1;  
$rs->("Batman")->("Robin");      # 将打印出 "Batman and Robin"  
  
sub test1 {  
    my $arg = shift;  
    print "$arg";  
    return \&test2;  
}  
  
sub test2 {  
    my $arg =shift;  
    print " and $arg\n";  
}
```

## 符号引用

回想一下，符号引用包含的只是名字（字符串），而不是真正的引用。真正的引用与符号引用间并没有语法上的差别，如：

```
sub foo { print "foo called\n"}
$rs="foo";
&$rs();                # 将打印出 "foo called"
```

## 使用子例程引用

让我们来看一些子例程引用的常见例子：回调函数与高等级子例程。

回调函数就是通过引用来使用的普通子例程。调用者（那些引用使用者）不必了解究竟是调用了哪一个子例程。让我们来看一下使用回调函数的三个简单例子：调度表、信号处理程序和绘图函数。

## 调度表

典型的调度表就是一个包含子例程引用的数组。下面的例子中，%options 就是一个将命令行选项映射到不同子例程的调度表：

```
%options = (          # 针对每个选项调用相应的子例程
    "-h"              => \&help,
    "-f"              => sub {$askNoQuestions=1},
    "-r"              => sub {$recursive =1},
    "-default_"       => \&default,
);

ProcessArgs(\@ARGV, \%options); # 以引用方式传递两个参数
```

这段代码中的引用有些指向有名的子例程，有些则由于做的工作不多而直接以匿名子例程的形式嵌入其中。现在我们可以将ProcessArgs编写成一种通用的代码。它使用两个参数，一个是用来进行解析的指向数组的引用，另一个是用于处理数组时将选项映射为子例程引用的散列表的引用。对于每个选项，它将调用相应的



“映射”函数，而且在 @ARGV 中遇到无效的选项时，调用与字符串 `_default_` 相对应的函数。

ProcessArgs 的代码在例 4-1 中。

例 4-1: ProcessArgs

```
ProcessArgs (\@ARGV, \%options);          # 两个参数均以引用方式传递
Sub ProcessArgs{
    # 注意记号的意义: rl 表示指向数组的引用, rh 表示指向散列表的引用
    my ($rlArgs, $rhOptions)=@_;
    foreach $arg (@$rlArgs){
        if (exists $rhOptions->{$arg}){
            # 其值一定是指向子例程的引用
            $rsub=$rhOptions{$arg};
            &$rsub()                        # 调用它
        }else {                            # 选项并不存在
            if (exists $rhOptions->{"_default_"}){
                &{$rhOptions{"_default_"}};
            }
        }
    }
}
```

你可以像下边这样使用块方式的间接访问来节省一步（请仔细查看以下第一章“数据表示与匿名存储”中的有关“一种更加通用的规则”一节）：

```
if (exists $rhOptions->{$arg}){
    &{$rhOptions->{$arg}}();              # 只用一步实现间接访问和调用子例程
}
```

我喜欢前面更“唠叨”一些的版本，因为它的可读性好。

## 信号处理程序

通常情况下，程序通过调用由操作系统实现的函数来工作，而不是反过来。但是当操作系统有紧急信息要发送给程序时，就会成为有悖于这条规则的例外。在许多操作系统中，这种信息传递是通过信号来实现的。例如，当用户按下 Ctrl-C 时，当硬件捕获浮点例外或当子进程退出时，都可能产生信号。你可以指定一个函数，

当程序接受到发送来的信号时由它来处理。这样可以使你采取相应的动作。例如，一个Ctrl-C句柄可以在退出前做一些清理工作。一个浮点例外句柄可能会设置一个错误标志，然后恢复正常的工作。

Perl提供了一种方便的方法来为每一种类型的信号指定信号处理程序。有一个名为%SIG的特殊变量，它的键值是信号名，而值为相应的子例程名或引用，根据相应的信号来调用相应的子例程。

```
Sub ctrl_c_handler {
    Print "Ctrl-C pressed \n";
}
$SIG{"INT"}=\&ctrl_c_handler;      #INT 表示中断信号
```

这里INT是一个特殊的字符串，它会根据键盘中断Ctrl-C来发送信号。操作系统的文档中列有发送到程序或脚本程序的信号的名字。实际上，你可以通过Perl打印出的设置信息来获取此类信息：

```
use Config; # 加载Config 模块
print $Config{sig_name};
```

当你赋值给%SIG时，你可以不必使用子例程的引用，因为Perl还允许你使用子例程的名字：

```
$SIG{"INT"}='ctrl_c_handler';      # 传递子例程名
```

在偶然情况下，信号处理程序非常危险。因为，Perl内部使用的C函数如malloc等是不可重入的。如果正在调用此类函数时触发了信号处理程序，而它又碰巧调用了这个函数，那么肯定会导致混乱而且还可能使程序崩溃。这种现象在脚本程序中更为隐蔽，因为你根本就不知道Perl会调用什么malloc函数。（第二十章“Perl的内部工作”将会向你阐明这个问题）。其中的经验就是在信号处理程序中做尽可能少的工作，如将先前定义的某个变量设置为真，然后在外面的代码中检查该变量的值。

## 表达式绘图

设想你要为具有如下形式的多个函数绘图：

```
y=f(x)
```

这里  $f(x)$  以一个数值为参数并返回另一个值。这种例子包括  $\sin(x)$ 、 $\cos(x)$  和  $\sqrt{x}$ 。而且我们还可以绘制如下所示任意复杂的表达式：

```
y=sin(2x)+cos2(x);
```

可以很容易的编写一个 `plot` 子例程，用它来在 0 到  $2\pi$  的区间中绘制这个表达式：

```
$PI = 3.1415927;
$rs = sub {                                # 匿名子例程
    my($x)=@_;
    return sin(2*$x)+cos($x)**2;          # 要绘制的函数
};
plot($rs, 0, 2*$PI, 0.01);
```

这是一个应用高等级子例程的例子，该子例程以一个用户定义的子例程(的引用)为输入参数并调用多次。`sort` 是一个内建高等级函数的例子；它与前者的区别就是，它需要的是子例程名而不是引用。

## 闭包

不仅可以返回数据，Perl 的子例程还可以返回一个指向子例程的引用。这同其他的传递子例程引用的方法没有什么不同。但是有一点是个例外，这是一种涉及匿名子例程和词法变量 (`my`) 的隐含特性。考虑下面的例子：

```
$greeting = "hello world";
$rs = sub{
    print $greeting;
};
&$rs();                                # 将打印出 "hello world"
```

在这个例子中，匿名子例程利用了全局变量 `$greeting`，没什么特别的，对吗？现在，让我们来稍微修改一下这个平实的例子：

```
sub generate_greeting{
    my ($greeting)="hello world";
```

```

    return sub{print $greeting};
}
$rs=generate_greeting();
&$rs();                # 打印结果为 "hello world"

```

generate\_greeting子例程返回一个指向匿名子例程的引用,而后使用该引用打印出 \$greeting。奇怪的是, \$greeting是一个从属于 generate\_greeting 的 my 变量。一旦 generate\_greeting 执行完,你会认为它所有的局部变量均被释放。但是在你后来又使用 &\$rs 调用匿名子例程时,它却仍然能够打印出 \$greeting 的内容。这是怎么一回事呢?

任何定义在匿名子例程中表达式当时就已经使用了 \$greeting。一个子例程块,从另外的角度讲,就是一块在以后某个时候调用的代码,因此它将保留所有要在以后使用的变量。(从某种意义上说就是把它带走。)当该子例程后来被调用并执行 print "\$greeting" 时,它会记得当初子例程创建时变量 \$greeting 的值。

让我们再来改动一下这段代码,以更好地理解闭包这个专业用语究竟能做什么:

```

sub generate_greeting("hello");
  my($greeting)=@_;          # 由参数初始化 $greeting
  return sub {
    my($subject)=@_;
    print "$greeting $subject\n";
  };
}
$rs1=generate_greeting("hello");
$rs2=generate_greeting("my fair");

# $rs1 与 $rs2 是两个分别保留不同 $greeting 值的子例程
&$rs1("world");              # 打印出 "hello world"
&$rs2("lady");                # 打印出 "my fair lady"

```

我们不是硬性的编写 \$greeting 信息,而是从 generate\_greeting 中获得。当 generate\_greeting 第一次被调用时,它返回的匿名子例程保留了 \$greeting 的值,因此 \$rs1 所指向的子例程就是完成了类似下面的这些功能:

```

$rs1 = sub{
  my ($subject)=@_;
  my $greeting="hello";

```

```
    print "$greeting $subject \n";          # $greeting 的值为 "hello"
}
```

该子例程就称之为闭包 (*closure*) (这个词来源于 LISP 语言)。正如你所看到的, 它捕获了 `$greeting` 的值, 而在后面被调用时, 它只需要一个参数就行了。

像某个外国移民保留着自己出生地的文化和风俗习惯一样, 闭包就是一些保留着在其创建时所在域中需要的变量。

Perl 创建的闭包恰好只针对 (`my`) 词法变量而不对全局或局部化 (使用 `local`) 变量。让我们来看一下这样做的原因。

## 闭包的内幕

如果你对闭包的工作原理不感兴趣的话, 你可以将本节跳过去而不会有什么不连贯的损失。

大家还记得变量名与变量的值是两个分立的实体。当第一次看到 `$greeting` 时, Perl 会将名字 “greeting” 同一个新分配的标量变量值进行联编, 并把值的引用计数设置为 1 (现在只有一个箭头指向该值)。在代码块结束时, Perl 将会分断名字与值的关系并将值的引用计数减一。在一个典型的代码块中, 如果你没有别的指向那个值的引用, 那么由于引用计数已经递减为零, 因此值将被释放。但是在上面的例子中, 碰巧匿名子例程要使用 `$greeting`, 于是使该变量的引用计数加 1, 这就防止了在 `generate_greeting` 结束以后被自动释放。当再一次调用时, 名字 `greeting` 则联编到另外一个新分配的标量变量值上, 于是第二个闭包则依附于它自己的标量变量值。

为什么对于 `local` 变量闭包就不能工作呢? 回想一下第三章 “Typeglob 和符号表”, 以 `local` 限定的变量是动态分配的 (或者称 “暂时的全局性”)。`local` 变量的值依赖于使用它时的调用栈。因此, 如果 `$greeting` 被声明为 `local`, Perl 将会在匿名子例程被调用时, 而不是在定义时来查找它的值。你可以用一个小测试来验证一下:

```
sub generate_greeting {
```

```
    local ($greeting)=@_;  
    return sub {  
        print "$greeting \n";  
    }  
}  
$rs=generate_greeting("hello");  
$greeting="Goodbye";  
&$rs();                #将打印出 "Goodbye", 而不是 "hello"
```

在这种情况下该匿名子例程不是闭包,因为它并没有保留其创建时\$greeting的局部值(“hello”)。一旦generate\_greeting结束执行,\$greeting又恢复到它先前的全局值,这个值也就是匿名子例程在执行时所看到的。

看起来好像generate\_greeting每次执行并返回匿名子例程时,都在内部创建一组新的代码。实际情况并不是这样。匿名子例程的代码只在编译时创建一次。\$rs在内部是一个指向一种“代码值”的引用,而“代码值”不但保留它们自己的字节码(这是在所有其他子例程引用中共享的),而且还有相应环境中所需要的所有变量(每个子例程都将自己的私有上下文环境打包供以后使用)。第二十章将更为切实地涉及更为确切的细节。

总的来说,一个闭包就是一种特殊的匿名子例程,它保留自己被创建时所在作用域中要使用的值。

## 闭包的应用

闭包以两种似乎毫不相关的方式来应用。最常见的用法就是用“智能”回调子例程(“smart” callback procedure)。另一个则是应用在一种称作“迭代器(iterator)”[或“流(stream)”,这是LISP中的概念]的专业用语中。

### 将闭包用作“智能”回调

由于闭包是一些包含少量私有数据的子例程引用,因此它们可以方便的应用在图形用户界面的回调过程中。

举例来说，你使用Tk工具包创建了一个按钮，并赋给它一个指向子例程的引用。当按钮被按下时将回调该子例程。现在如果将该子例程分配给两个不同的按钮，问题就出来了，子例程怎么会知道是哪一个按钮在调用它呢？其实很简单，你可以分配给它一个“智能”回调子例程——一个闭包。该闭包会保存一些按钮特定的数据，于是当子例程被调用时，

它可以魔术般的存取那些数据，如例4-2所示。

这个例子创建两个按钮，当被按下时会打印出它们的标题字符串。尽管这些关于包，特别是Tk模块的讨论还要在后面的章节中讲到，你还是能理解例4-2代码中的关键部分。此刻你应该将注意力放在使用闭包的部分上，不要考虑Tk模块的机制。

CreateButton创建了一个GUI按钮，并提供给它一个指向匿名子例程的引用（\$callback\_proc），它将依附于其环境中的my变量\$title。当用户按下按钮时，回调过程将被调用，并使用它所保存的\$title的值。

例4-2：传递闭包而不是普通的子例程

```
use Tk;
# 创建顶层窗口
$stopwindow=MainWindow->new();
# 创建两个按钮，这些按钮将在被按下时打印自己的名字
CreateButton($stopwindow, "hello");
CreateButton($stopwindow, "world");
Tk::Mainloop();      # 开始事件分发
#-----
sub CreateButton{
    my ($parent, $title) = @_;
    my ($b);
    $callback_proc = sub{
        print "Button $title pressed \n";
    };
    $b = $parent->Button(
        '-text'    => "title",      # 按钮标题
        '-fg'      => 'red',        # 前景色
        '-command'=> $callback_proc # 当按钮按下时调用的子例程
    );
    $b->pack();
}
```

注意，按钮本身并不关心它们获得的是对普通子例程的引用还是闭包。

## 迭代器与流

迭代器跟踪保留它在一串实体“流”中的当前位置，并在下次被调用时返回下一个逻辑实体。它有点像数据库的游标，游标返回的是记录流中的下一条记录（匹配某个查询的所有记录）。流可以是有界的（如一组数据库记录），也可以是无界的（如偶数流）。

让我们来看一下如何用闭包来表示流和迭代器。第一个例子是一组偶数流和作用其上的迭代器，它在每次调用时返回下一个偶数。很显然，我们无法生成所有可能的偶数（这是个有界的例子），但是我们却总是可以在记住前一个偶数的情况下，计算出下一个偶数。迭代器将为你记住这种关键信息。

子例程 `even_number_printer_gen` 以一个整数为参数，返回用于打印出起始于给定整数的偶数的子例程（注 1）。该程序如例 4-3 所示。

例 4-3：一个偶数流发生器

```
sub even_number_printer_gen{
    # 该函数返回指向匿名子例程的引用
    # 该匿名子例程打印出从 $input 开始的偶数
    my ($input) = @_;
    if ($input % 2){$input++;           # 如果给定的数为奇数，则计算出下一个偶数
    my $rs=sub {
        print "$input";                 # $input 是在外围定义的 my 变量
        $input +=2;
    };
    return $rs;                         # 返回指向上面子例程的引用
}
```

下面是如何使用：

```
# 你需要从 30 开始的偶数。通过 even_number_printer_gen 来产生定制的完成此类工作的迭代器
```

注 1： 这个例子和讲解是以 Robert Wilensky 的优秀图书《LISPcraft》（W.W.Norton 公司出版）为基础的。



```

$iterator=even_number_printer_gen(30);
# $iterator 现在指向一个 CLOSURE
# 你每次调用它时都将打印出下一个连续的偶数
for ($i=0; $i<10;$i++){
    &$iterator();
}
print "\n";

```

打印结果为:

```
30 32 34 36 38 40 42 44 46 48
```

\$iterator 保留着 \$input 的值，并作为以后操作中的私有存储来保存下一个偶数。当然你可以创建任意多的迭代器，每个迭代器都以自己的起始数值初始化：

```

$iterator1=even_number_printer_gen(102);
$iterator2=even_number_printer_gen(22);

&$iterator1();      # 打印 102
&$iterator2();      # 打印 22
&$iterator1();      # 打印 104
&$iterator2();      # 打印 24

```

要注意每个子例程都使用自己的私有 \$input 的值。

两个闭包可以共享变量吗？当然可以，只要这两个闭包在同一个环境下创建。例 4-4 产生两个匿名子例程，一个打印偶数另一个打印奇数。它们都紧接着上一个数字打印（打印出奇数或偶数），而不管任一个函数在过程中被调用了多少次。

例 4-4：闭包的共享变量

```

sub even_odd_print_gen {
    # 这两个过程共享 $last
    my ($rs1, $rs2);
    my ($last) = shift;      # 它被下面的两个闭包共享
    $rs1=sub {               # 偶数打印机
        if ($last%2){$last+=2}
        else{$last++};
        print "$last \n";
    };
    return ($rs1, $rs2);     # 返回两个匿名子例程的引用
}

```

```
($seven_iter, $odd_iter)=even_odd_print_gen(10);  
&$seven_iter();           #打印 12  
&$odd_iter();              #打印 13  
&$odd_iter();              #打印 15  
&$seven_iter();            #打印 16  
&$odd_iter();              #打印 17
```

这个例子利用了Perl可以从一个子例程中返回多个值的功能，因此返回两个匿名子例程的引用不成问题。而这两个子例程恰巧都访问`$last`。你可以使用不同的初始值调用`even_odd_print_gen`任意次，而它每次都将返回一对子例程闭包。重要一点是，要想共享数据，匿名子例程必须在相同的作用域中创建。本例还强调了这样一个事实，那就是闭包的确与它所需要的`my`变量直接联编，而不是拷贝或计算变量的值。

## 随机数生成

让我们将注意力转向一个更为有用的例子，是有关无界流，也就是随机数流的。我们使用与前面的例子相同的策略：由迭代器跟踪保留上一个产生的伪随机数。

你或许会争论说，`rand()`函数不就表示以一个种子（使用`srand`产生）为初始化值的迭代器吗。不错，但是如果我们要编写一个依赖于两个独立随机数生成器的模拟程序呢。在这两个生成器中都使用`rand`并不能保证它们相互独立。因为`rand`碰巧要使用上次产生的随机数（保存在全局变量中）来计算出下一个新的随机数。因此对一个流调用`rand`会影响到另一个流所产生的数字。

闭包则提供了一种很好的解决方案，因为它们本身就是一种代码与私有数据的结合。我们不使用`srand`，而是使用`my_srand`，它将返回一个以恰当的数值为种子值的定制的随机数生成子例程。换句话说就是，`my_srand`是一种“随机数生成器的生成器”，由它返回一个以`$srand`为初始值的定制的匿名子例程。

例4-5为实现代码，请不要太注意算法本身（线性重叠算法），因为对于所选择的特定常量的随机性还没有经过测试（而且它每1000个数字要重复一次）。况且，还有更好的算法。

例 4-5: 一个随机数生成流

```
sub my_srand {  
    my ($seed)=@_  
    # 它返回一个随机数生成器函数  
    # 它是可预测的, 该算法要求你提供一个随机的初始值  
  
    my $rand=$seed;  
    return sub {  
        # 根据以前的值计算出新的伪随机数  
        # 数字的区间为 0 到 1000  
        $rand = ($rand*21+1)%1000;  
    }  
}
```

我们现在就可以任意次地调用 `my_srand`, 并获得完全相互独立的闭包, 而每个闭包又能分别从它个自的起点生成随机数:

```
&$random_iter1 = my_srand(100);  
&$random_iter2 = my_srand(1099);  
for ($i=0;$i<100;$i++){  
    print $random_iter1(), " ", $random_iter2(), "\n";  
}
```

## 闭包与对象的对比

如果你没有面向对象的背景知识, 你也许在阅读了第七章“面向对象编程”之后才能更好的理解这一节的内容。

对象通俗的定义就是一组数据与函数的集合。数据提供对象函数操作的环境。比如, `$button->setForeground("yellow")` 中, `setForeground` 函数会自动知道你指的是哪个按钮。

从某种意义上说, 闭包也提供了同样的机制——它也是一种子例程与只对子例程可见的私有数据的结合。在我们前面举的例 4-4 的 `even_odd_print_gen` 中, 可以有任意数量的子例程来访问相同的数据, 前提是它们都必须在相同的作用域中创建。Abelson、Sussman 和 Sussman 所著的一本不错的名为《Structure and

Interpretation of Computer Programs》的书中描述了在 Scheme（一个 LISP 的变种）中如何创建和使用此类对象。

Perl 支持许多面向对象的特性（如类似 C++ 中的继承和虚函数），这使它更容易的以面向对象的风格而不是使用闭包来创建迭代器和流（对象的属性反映迭代器的状态）。闭包比起对象来，要花费更多的存储空间但速度却稍微快一些。我们将在第二十章研究这样的原因。

我更倾向于使用对象而不是闭包，但是有一种例外情况：那就是回调过程。我发现使用简单的闭包创建回调要比创建回调对象（这是你在 C++ 中的常用方式，在 Java 中则是唯一的方式）更容易。在上面 Create-Button 的例子中，你可以只使用一个“方法”就能够创建一个回调对象，比如 `execute()`。当点击按钮时，方法 `$callback_object->execute()` 将被调用，而且那个对象的 `execute` 方法，将会确切知道去做什么。回调对象将保存 `execute` 工作的环境。上面这么多工作，如果使用闭包则会更为简单和直接，因为它自动保留所需的环境。

在 Tom Christiansen 的 *perltoot*（*toot* 表示 Tom 的面向对象指导教程）文档中，就实现了利用闭包来表示状态的对象。这是一种有趣的方式，但是由于存在保证数据隐藏的更简单和快速的方法，因此我没有去使用它。我们将在第七章更多的谈到这里的内容。

## 和其他语言的比较

### Tcl

Tcl 程序员大量的依赖动态数值计算（使用 `eval`）来传递各类代码。虽然你也可以在 Perl 中这么做，但是 Perl 的匿名子例程是一些编译好的代码，它执行起来当然要比动态数值计算快许多。Perl 的闭包还提供了 Tcl 中不存在的其他优点：在不同闭包间共享私有数据的能力（在 Tcl 中，要想共享就必须为全局变量）。而且不用担心变量的替换规则。（在 Tcl 中，你必须认真的亲自使用替换规则展开所有的变量，而后才能将一段代码传递给其他人。）

## Python

Python提供了一种能力较低的闭包形式：一个子例程只能使用离它最近的包含环境中的变量。这被称做“浅联编”，而Perl提供的是“深度联编”。Mark Lutz的书《Programming Python》（O'Reilly & Associates 公司 1996 年出版）中提供了一种达成深度编联的迂回解决方案，它是通过将其毗邻作用域中的值设置为默认参数来完成的。

我还是更喜欢 Perl 这样的环境，能为我自动的处理这些工作。

## C++

C++支持指向子例程的指针但并不支持闭包。你必须在回调例程需要操作作用的上下文数据时使用这种所谓的回调对象。如果你不想再创建另外的回调对象，你还可以从标准的回调类中继承的方式来获得这种对象，然后提供一个重载方法，如“execute”，这样，调用者就可以简单的使用 `callback_object->execute()` 来完成相应的工作了。

## Java

Java语言既不提供闭包也不提供指向子例程的指针（方法）。我们可以使用接口来提供一种标准化的回调机制，这样调用者就不必关心产生对象的特定的类了（前提是要实现那个接口）。

## 相关资源

1. perlref, perlmod, perlsub, perltoot（Perl 文档）。
2. Structure and Interpretation of Computer Programs. Harold Abelson, Gerald Jay Sussman, Julie Sussman. MIT Press, 1996。

使用 LISP 来解释高等级过程和闭包，值得一读。

# 第五章

## Eval

### 本章简介:

- 字符串形式: 表达式计算
- 代码块形式: 例外处理
- 注意你的引号
- 使用 Eval 进行表达式计算
- 为了效率而使用 Eval
- 使用 Eval 来处理超时
- 其他语言中的 Eval
- 相关资源

一个人的数据就是另一个人的程序。

—— Jon Bentley

《Programming Pearls》

几年前, 我的一个朋友向我展示了一个精巧的运行在微小的 32K 机器 (BBC Micro) 上的程序, 它可以接收任意的诸如  $\sin(x) + \cos(x*2)$  的表达式并绘制相应的图形。由于以前没怎么学过语法分析器, 我当时就感到很奇怪, 这东西到底让他写了多少行代码。他让我看了看程序; 整个程序竟然一屏就能够显示完。他使用了 BASIC 语言提供的 eval 语句。

大多数像样的脚本语言, 如 BASIC (只有其中的一些版本), Perl、Tcl、LISP 和 Python 等都有一种特性, 这种特性将它们同类似 C 这样的系统编程语言区分开来。这就是将字符串当作小程序的能力 (注 1)。

对我来说, Perl 的运行时数值计算能力是我选择使用它的最重要的原因之一。(另一个就是对正则表达式的强大支持。) 我使用运行时数值计算来直接创建小代码片段, 然后再交给 Perl 快速执行 (速度很快!), 用以创建复杂的小型语言解释器 (注 2)。eval 函数是通向这种强大功能的大门。特别的, 我们将在第七章“面向

注 1: 请见附录二中与之有关的“动态特性”一节的内容, 来了解将 Perl 与系统编程语言区分开来的其他 Perl 结构。

注 2: 想要一些有关小语言的令人愉快的讨论, 一定要看一下 Jon Bentley 的《More Programming Pearls》一书。

对象编程”中用它来创建对象属性的存取函数，在第十一章“对象持续性的实现”中，用它来创建 SQL 查询计算器。

正如我们所要看到的，Perl 根据其参数类型的不同，以两种似乎截然不同的方式工作。如果给定的是字符串，Perl 就会将它当作小程序并编译执行（如前面所提到的那样）；这被称做动态表达式计算。字符串的内容在编译时可以是已知的也可能是未知的。如果给定的参数为一个代码块——也就是说在编译时代码是知道的，则 eval 将被用来捕获运行时例外。

动态表达式计算与例外处理是两种非常不同的概念，同时大家也希望能够使用两种不同的关键词来完成各自的任务。Larry Wall 有一次提到，他曾经考虑过使用不同的关键词 try，用来做例外处理。但是他在那个时候正保守的使用关键词。我觉得这个词就挺好，因为动态表达式计算很可能就会产生像编译代码那样的运行时例外。

在本章中你将深刻地理解这两种形式的 eval 是如何工作的，同时也将在你使用的专业用语中增加重要的一项。

## 字符串形式：表达式计算

当给定 Perl 一个文件，或以命令行的方式（使用 -e）执行程序时，它需要对程序内容进行语法分析，查找语法错误并在一切都正常时再执行它。Perl 通过“eval 字符串”的形式使这项功能对程序员来说也能使用。这一点使 Perl 同诸如 C、C++ 或 Java 语言形成巨大反差。对于后者编译器本身与你的程序是分立的实体，在运行时是不可用的。换句话说，Perl 解释器自身完成的工作多少有点类似下面的情形：

```
# 一口气读进整个文件的内容
while ($line = <>){
    $str .= $line;           # 将所有行连接起来
}

# 现在 $str 中包含有整个文件的内容。执行它!
eval $str;
```



如你所见，eval 将处理提交给它的任意的 Perl 脚本程序。更棒的是这种机制不仅对 Larry 可用，而且对你和我这样的人同样可用。请尝试下面的代码：

```
# 在 $str 中放入一些代码
$str = '$c=$a+$b';           # Perl 并不关心你在 $str 中都放了些什么
$a=10; $b=20;
eval $str;                   # 把 $str 当作代码执行
print $c;                    # 打印结果为 30
```

在这段代码中，\$str 先是被当作普通的字符串，填进去什么就是什么。但是 eval 则把它当作程序并加以执行。重要的一点是，eval 不把它当作一个另外的程序，而就像去掉 eval 语句后直接嵌在原来的程序中一样，如图 5-1 所示。

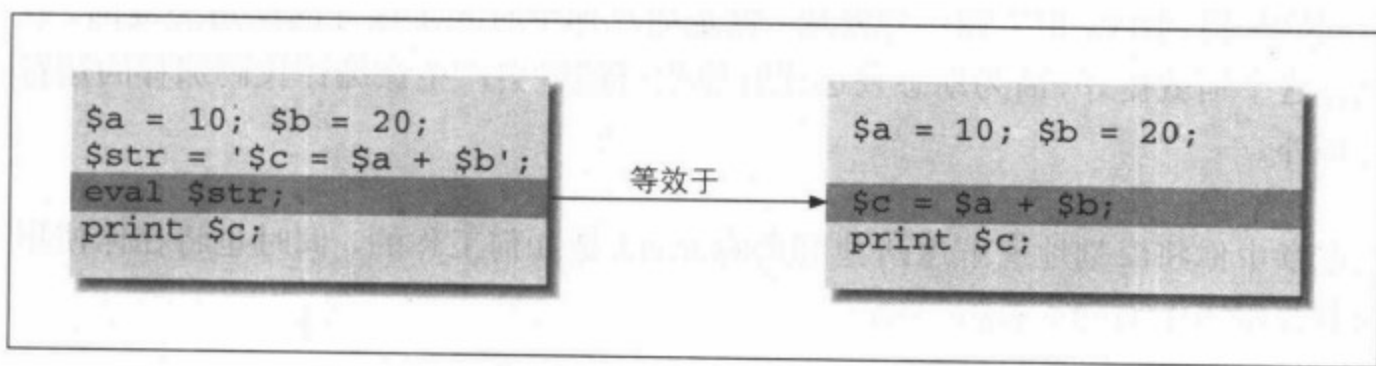


图 5-1 eval 在它自己的上下文环境中编译执行字符串

因此，作为 eval 参数的字符串可以使用在那个上下文中的变量和子例程，这包括 my 和 local 变量，而且可以选择在其中创建新的变量和子例程。在前面的例子中，提供给 eval 的字符串将两个经过初始化的变量相加，并产生一个新的变量 \$c。

如果你在字符串中包含不止一条语句（记住，字符串可以像你所需要的程序一样大），eval 将计算所有的语句并返回最后一条语句的值：

```
$str = '$a++; $a+$b';        # 包含两条语句
$a=10; $b=20;
$c=eval $str;                # $c 的值为 31（这是第二条语句的值，$a+$b）
```

当然，像上面的例子这样，使用 eval 语句来计算编译时就知道的代码，没有什么意义。当 \$str 的内容来自别的地方——如标准输入、一个文件或从网络上获得时，才会更有意义。我们就要看到一些这样的例子。



**注意：** 字符形式的 eval 是一种安全隐患。如果字符串的内容来自一个不可信的地方，包含了如下的句子：

```
system('rm *')
```

那么代码会顺利地得以执行，最终结果可想而知。当你不能信任输入的内容时，就可以利用 Perl 提供的污染检测选项，它可以防止从外界程序导入的数据影响该程序以外的文件或其他内容。你也可以使用 Perl 发行版所带的 Safe 模块，它可以为 eval 语句提供类似于浏览器中为 Java 或 Tcl/Tk 的小应用程序（applet）所提供的隔离保护。

如果 \$str 中包含了无效的表达式，会出现什么情况呢？那样的话 Perl 会将错误信息放入一个称做 \$@ 的变量中（如果你使用 English 模块，则是 \$EVAL\_ERROR）。由于在执行字符串之前要先进行编译，因此该错误可能是编译错误也可能是运行时错误。如果 \$str 中的代码准确无误（当然，我的意思是语法错，因为不可能保证你不出逻辑错误），则 \$@ 的值必定为 undef。

因为 Perl 本身也使用 eval 解析和执行脚本代码，因此该错误信息（在 \$@ 中）同你在使用 Perl 处理有错误的脚本时，在标准错误输出中显示的字符串一样。

还有一件微妙但很重要的事要说明。eval 把该字符串当作一个代码块，这也是它能够处理多条语句（而不仅仅是表达式）并返回最后一条语句的值的值的原因。也就意味着，你无法看到对 eval 字符串中的局部和词法变量的改变。

## 代码块形式：例外处理

在这种形式下，eval 后面跟的是一个代码块，而不再是包含字符串的标量变量。它被用来处理运行时错误，或称做例外。错误可以是内建的（如内存溢出，除数为零），或者是用户通过使用 die 来自己定义的。

下面的例子向你展示了如何通过使用代码块方式的 eval，来捕获除数为零的运行时错误：

```
eval{
    $a=10;$b=0;
    $c=$a/$b;      # 将导致由 eval 捕获的运行时错误
};
print $@;          # 将打印出 "Illegal division by 0 at try.pl line 3"
```

在编译脚本时，Perl 对代码块进行语法检查并生成编译代码。在遇到运行错时，Perl 将跳过 eval 块中剩余的代码，并将 \$@ 设置为相应的错误信息。

为了产生自己的错误，你需要使用 die。Perl 知道某一段代码是否在 eval 块中执行，因此，当 die 被调用时，Perl 只是简单的将错误信息赋值给全局变量 \$@，并跳转到紧跟 eval 块的语句继续执行。在下面的例子中，open\_file 在无法打开文件时调用 die。将它包裹在 eval 中来使用这个函数。

```
sub open_file{
    open(F, $_[0]) || die "Could not open file $!";
}

$f='test.dat';
while(1) {
    eval{
        open_file($f);           # 如果 open_file 出错，程序将不会退出
    };
    last unless $@;             # 如果没有错误，就退出循环
    print "$f is not present. Please enter new filename $f";
    chomp($f=<STDIN>);
}
```

Java/C++ 程序员肯定认出了它们与 throw、try 和 catch 语句的相似之处，try 对应 eval 块，catch 对应对 \$@ 的检查，而 throw 对应 die。[本质上相当于，调用者对运行环境说，“来，try（试）一下这段代码，并 catch（捕捉）任何一种由被调用者 throw（抛出）的错误。”]

在 C++ 和 Java 中，一个函数可以重新抛出自己不想处理的例外。在 Perl 中你可以通过调用不带参数的 die 来达到上述目的。

```
eval {
    ...
};
if ($@ =~ /sorry, bucko/){
    ...
} else{
    # 哦，我不知道它怎么处理
    die;                      # 等价于 die $@
}
```

如果存在包含上面这段代码的 eval 块，那么该例外将被捕获，否则程序终止。

## 标准模块

由于 C++ 和 Java 中包含有用于捕获和处理错误的特殊结构，因此一些 Perl 程序员也想这么做。下面是一些选择方案。

### Exception.pm

当这本书出版时，有一个新的名为 Exception 的模块刚刚在 CPAN 上发布，它是建立在 eval 和 die 之上的。要理解下面的例子，你得理解 Perl 对面向对象的支持，所以你需要在看完后续章节后再重新来看看这个例子。

当你试图提取超过 \$300 或超出当前余额时，这段代码将会产生例外：

```
use Exception;
package AmountExceededException;          # 用户定义的例外
@ISA=('Exception');

package OverdraftException;               # 用户定义的例外
@ISA=('Exception');

package BankAccount;
sub withdraw_money{
    my $amount=shift;
    if($amount>300){
        throw new AmountExceededException;
    }
    if($amount > $balance){
        throw new OverdraftException;
    }
    ...
}                                           # 改变余额

try {
    print "How much do you need?"; chomp($amount=<STDIN>);
    withdraw_money($amount);
}
catch AmountExceededException =>
```

```
        sub { print 'Cannot withdraw more than $300'},
OverdraftException =>
        Sub {print $_[0]->message},
Default =>
        Sub {print "Internal error. Try later"};
```

## Exception.pl

在当前的标准 Perl 库中，有一个名为 *exception.pl* 的模块，它也是通过一层薄薄的包裹在 `eval` 和 `die` 之外的代码层，来提供名为 `catch` 和 `throw` 的子例程的。`catch` 的参数是包含代码的字符串（同前面的例子不同，它不是代码块），和若干在 `eval` 代码后与错误字符串进行匹配的正则表达式。

该模块有一个严重的问题，因为 `catch` 是子例程，所以当前域中的词法变量（使用 `my` 进行局部化）对它来说不可见。这个问题在更新的 *Exception.pm* 模块中已经得到解决。

我怀疑程序员们会鄙视那种将一种语言弄得像另外一种语言的做法（注3），而且归根结底，原始方式的 `eval` 和 `die` 或许是最简单的选择。

## 注意你的引号

Perl 解释器对引号和代码块的解释有不少微妙之处，考虑下面语句间的不同：

```
$str='$c=10';
# eval $str;           #1
eval "$str"            #2
eval '$str'            #3
eval { $str };         #4
```

第1种与第2种情况的结果完全相同，而第3和第4种之间则存在差异。你能说出为什么吗？技巧就是要知道在把字符串提交给 `eval` 之前，Perl 解释器都做了些什么。

---

注3： 这不包括 Larry Wall，想想看，他把 Perl 设计的有点儿像 C，有点儿像 sh，还有点儿像 awk！

在第1种情况下, Perl将\$str的内容提交给eval, 就像同对待其他函数一样。因此eval将字符串看作'\$c=10', 并当成一个小程序来执行。

在第2种情况下, Perl在提交给eval之前, 首先对双引号中的变量进行替换, eval同样可以看到\$str的内容, 于是就进行编译执行, 将10赋值给变量\$c。

在第3种情况下, 提交给eval的是由单引号括起来的字符串, 它在变量替换阶段是不会被展开的。因此, eval看到的仅仅是一个如此编写的字符串(包含有字符串"\$", "s", "t", "r"), 但eval还是像以前一样把它当作一个小程序。当然作为一个独立的程序, 它一点用也没有。由于eval要返回最后一个表达式的结果, 因此它就返回了\$str的值(也就是字符串\$c=10)。换句话说, 如果你这么写:

```
$s=eval '$str';
```

\$s的结果将包含\$c=10。

第4种情况与第3种情况相同, 只是要在编译时对代码块中的代码进行语法检查(与对其余代码检查的时机一样)。

eval的东西就这么多。现在我们来看一下如何应用它来进行表达式计算、例外处理和提高效率。

## 应用 Eval 来进行表达式计算

有许多诸如语法分析和表达式计算的烦琐工作, 你可以交给Perl来完成。前提当然是你的语法分析要求与Perl自身的类似。毕竟, Perl知道如何去解析和计算Perl风格的语句。

让我们假定你的输入数据是一组用引号括起来的字符串, 而且你想验证一下其中的引号是否配对:

```
'He said, "come on over"'
'There are times when "Peter" doesn\'t work at all'
```

你不必绞尽脑汁寻思反斜杠换码的问题或如何编写程序来检查引号是否配对(平衡),你只需像例5-1那样简单地eval那个字符串即可。要知道字符串也是一种正确的Perl表达式。如Perl在\$@中放置了错误信息,就可以肯定你的输入有错。

例5-1: eval.pl

```
while (defined($s=<>)){           # 读一行到 $s 中
    $result=eval $s;               # 对那一行进行计算
    if ($@){                       # 检查编译或运行时错误
        print "Invalid string: \n $s";
    }else{
        print $result, "\n";
    }
}
```

这段代码比较棒的地方还在于,它还可以工作得像一个优秀的计算器,这是因为 \$s 可以是任意有效的Perl语句,可以带有算术操作符、循环、变量赋值、子例程等等。下面演示如何使用这个程序:

```
% perl eval.pl
2 * log(10);
4.60517018598809
$a = 10;$a += $a ** 2;
110
for (1..10){print $_, " "}
1 2 3 4 5 6 7 8 9 10
```

你每输入一行,Perl都会计算并打印出结果(那些以粗体显示的内容)。对于一个如此简单的shell程序,你还索求什么呢?要注意的是,以上代码要求你每一行输入都必须是一条完整的表达式,因此你不能够书写多行表达式。但是你仍然可以改动一下程序,使你在输入一个空白行时开始进行计算。

值得将这几行代码同Kernighan和Pike在经典的《The Unix Programming Environment》一书中通过yacc、lex和C编写的计算器对比一下,这还不包括学习lex和yacc所花费的精力。其他静态语言如Java和C++也拥有类似强大的功能:由于你不能存取编译器本身的强大功能,因此你不得不从头写起。

## 替换中的表达式计算

Perl的替换操作符一般具有以下形式: `s/regex/replacement/`, 并且将输入字符串中任何与正则表达式模式匹配的地方替换为 `replacement`。/e 标志将会对这种替换加以修正: 它会告诉替换操作符后半部分是一个 Perl 表达式, 而不是普通的替换字符串; 表达式的结果才供替换使用。考虑下面的例子:

```
$line = 'Expression Evaluation';
$line =~ s/(\w+)/ scalar(reverse($1))/eg;
print $line ;                # 打印结果为 "noisserpxE noitaulavE"
```

替换操作符的第二个参数是一个表达式: `reverse` 被用在标量变量环境中将提供的字符串逆序排列。/g 标志确保每一个匹配的单词均被逆序排列。

这个话题似乎对关键词 `eval` 来说有点跑题, 但它仍然属于运行时表达式计算的范畴; 实际上, /e 就表示 “expression”, 而不是 “eval”。该表达式将在编译时进行语法检查, 因此如果你需要查看运行时错误的话, 你还是需要把整个语句放在 `eval` 块中。考虑下一个例子, 它将包含 “数字/数字” 模式的字符串替换为等价的分数:

```
$l='His chances of winning are between 2/5 and 1/3';
eval {
    $l =~ s/(\d+)/(\d+)/$1/$2/eg;
};
print $l unless $@;
```

打印结果为 “His chance of winning are between 0.4 and 0.333333333333.”。该 `eval` 块可以捕获除数为零的错误。

## 应用 Eval 来提高运行效率

下面列举了一些应用运行时计算来极大的提高执行速度的例子。

## 一种快速的多模式 grep

考虑一个类似于 *grep* 的 Perl 脚本程序，它能够查找任意多的模式并打印出匹配所有模式的行（模式的先后顺序并不重要）。程序的代码结构如下所示：

```
while ($s=<>){
    $all_matched=1;          # 开始就假定所有模式都匹配 $s
    foreach $pat (@patterns){
        if ($s !~/$pat/){
            $all_matched=0;    # 哦，我们的假设错了
            last;
        }
    }
    print $s if $all_matched;
}
```

这段代码的问题在于该正则表达式（`/ $pat /`）对于每一行，每一种模式都要重新编译。也就是说，假设你有 10000 行文本需要搜索，三个要搜索的模式分别为，`a.*b`、`[-9]` 和 `[^def]`，那么模式则要被编译 30000 次。用于告诉 Perl 进行正则表达式编译的 `/o` 标志不能用在里，因为 `$pat` 在程序执行时是变化的。

最快速的方案就是像下面这样将模式硬性的编制到代码中。不幸的是，这是重用性最差的一种方案。

```
while ($s=<>){
    if ( ($s~/a.*b/)&&
          ($s~/[0-9]$/)&&
          ($s~/[^def]/) ){
        print $s;
    }
}
```

有一个好办法，既能获得与上面同等的效率又不失通用性。这就是通过在运行时手工编制上面那种硬联编的代码，然后使用 `eval` 进行计算。

用于生成代码的字符串在例 5-2 中以黑体表示。

例 5-2：通过编译正则表达式字符串来获得高速度

```
$code='while (<>){'
```



```

$code.='if (/';
$code.=join('/&&/', @patterns);
$code.='/) {print $_;}}';
print $code, "\n";
eval $code;                # 噢，最后开始执行
# 检查输入模式中错误的正则表达式
die "Error ---: $@\n Code:\n$code\n" if ($@);

```

例如，如果@patterns中包含“^foo”、“bar\$”、“ghi”三个字符串，那么提供给eval的代码如下所示：

```
while (<>) {if (/^foo/&&/bar$/&&/ghi/){print $_;}}
```

将模式进行排序，把那些包含有行起始符或终结符（^或\$）的行先进行匹配，则可以使这个例子更高效。因为搜索位于行起始或终结的模式，要比搜索整个字符串要快得多。另一项改进就是让用户来提供布尔操作符，而不是把&&硬编码到程序中。你可以查看一下Perl发行版中所带的*perlfaq6*文档（有关正则表达式的FAQ）。

## 从文件中获取相应的列

让我们来看看另一个用以提高运行效率，动态创建并计算Perl代码的例子。我们编写了一个名为col的程序，它可以从文件中获取相应的列，这有点类似于Unix的cut(1)命令。它的使用方法如下：

```
%col -s80 8-14 20+8 test.dat
```

上面的程序将*test.dat*当作一个固定格式的文件，每条记录为80列，程序将从每条记录中提取两列，第一列从第8个字符位置开始到第14个为止（最左边的列的索引为1而不是0），另一列从第20到第28，如图5-2所示。如果没有给定-s选项，程序将会把换行符当作记录结束符并一行一行的读文件。col允许列区间的重叠。

你已经知道，substr可以在给定起始位置和子串长度的情况下提取子字符串。通过在循环中（每个区间循环一次）调用substr的方法来编写col程序，是件简单的工作：

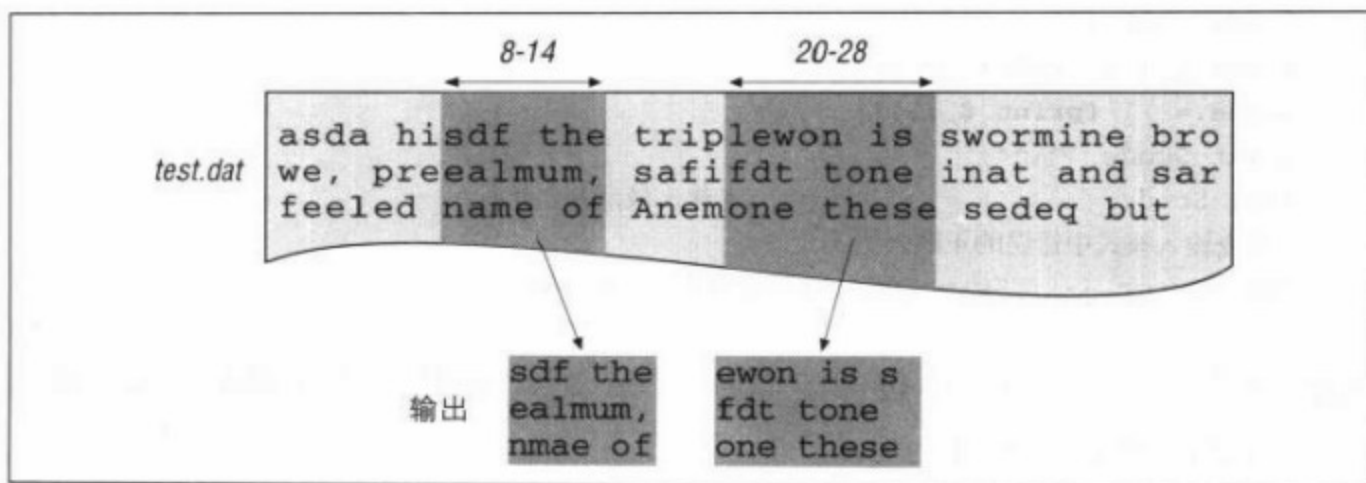


图 5-2 使用 col 来提取相应的列

```

for each line in the file {
    for each column range in the command line arguments{
        print substr(line, range);
    }
}

```

另外,我们不使用unpack来代替substr,这是因为我们希望输入区间可以重叠。

比前者更为高效的另一种方式就是“展开循环”,并且尽可能的使用常量,如下所示(这是对于上面的命令行调用来说的),对每条从文件中读出的记录,这段代码将根据输入区间提取子串,并适当的添加空格符。它还在每个提取的列间增加列分隔符(“|”)。

```

#PART 1 -----
sub col {
    my $tmp;
    while(1){
        $s=get_next_line();
        $col="";

#PART 2 -----
        $s.=' 'x(14-length($s)) if (length($s)<14);
        $tmp=substr($s, 7, 7);
        $tmp.=' 'x(7-length($tmp));
        $col.='|' . $tmp;
        $s.=' 'x (28-length($s)) if (length($s)<(28));
        $tmp=substr($s, 19, 9);
        $tmp.=' 'x(9-length($tmp));
    }
}

```

```

        $col.='|' . $tmp;

#PART 3 -----
        print $col, "\n";
    }
}

```

\$tmp在任何时候都只包含一列的数据,而\$col则用来累积提取的每列数据最后再将它打印出来。

给出所示的命令行参数,让我们来在运行时构造该子例程。注意,第1部分与第3部分是独立于命令行参数的,而只有用于提取行中所有列的第2部分才与命令行参数相关。

我们以前提到过,你必须非常留意引号的使用。假定\$col1和\$offset中分别包含7和6,这样我们就需要最终将下面的代码插入到我们的可执行字符串中:

```
$tmp=substr($s, 7, 6);
```

我们是这么来产生这行代码的:

```
$code = '&tmp=substr($s, ' . "$col1, $offset)";
```

这里我们通过使用单引号和双引号来仔细的控制变量替换。例5-3列举了用于生成三部分代码的三个子例程generate\_part1, generate\_part2和generate\_part3。子例程get\_next\_line将制表符转换成等价的空格符,以保证制表符的视觉效果。generate\_part3在完成col代码的同时还执行了生成的代码。和往常一样,我们用黑体来表示代表代码的字符串。

b

例5-3: col: 从文件中抽取相应列的脚本程序

```

# 从文件中抽取相应列的文本
# 用法: col [-s<n>] col-range1, col-range2, files ...
# 这里 col-range 按照 col1-col2 (从列1到列2)
#      或 col1+n的方式指定, n是列数
$size = 0;                # 0 表示面向行的输入, 否则就表示固定格式
@files = ();              # 文件列表
$open_new_file = 1;       # 要求 get_next_line() 打开第一个文件
$debugging = 0;           # 使用命令行标志 -d command 来激活

```

```

$col = "";
$code = "";
generate_part1();
generate_part2();
generate_part3();
col();          # sub col 现在已经被产生出来了，接着就调用它!
exit(0);

#-----
sub generate_part1 {
    # 生成 sub col() 中起初固定不变的代码
    $code = 'sub col { my $tmp;';          # 注意单引号的使用
    $code .= 'while (1) {$s = get_next_line(); $col = "";';
    $delimiter = '|';
}

#-----
sub generate_part2 {
    # 参数处理
    my ($col1, $col2);
    foreach $arg (@ARGV) {
        if (($col1, $col2) = ($arg =~ /^(\d+)-(\d+)/)) {
            $col1--; # 使其基于0
            $offset = $col2 - $col1;
            add_range($col1, $offset);
        } elsif (($col1, $offset) = ($arg =~ /^(\d+)\+(\d+)/)) {
            $col1--;
            add_range($col1, $offset);
        } elsif ($size = ($arg =~ /^-s(\d+)/)) {
            # 空操作
        } elsif ($arg =~ /^-d/) {
            $debugging = 1;
        } else {
            # 必须是一个文件名
            push (@files, $arg);
        }
    }
}

#-----
sub generate_part3 {
    $code .= 'print $col, "\n";}}';

    print $code if $debugging; # -d flag enables debugging.
}

```

```

    eval $code;
    if ($@) {
        die "Error ..... \n $@\n $code \n";
    }
}

#-----
sub add_range {
    my ($coll, $numChars) = @_;
    # 如果检索超出了字符串的尾部, substr 就会报错 (在设置了 -w 的情况下)
    # 为了避免这种情况的发生, 在必要时为字符串补充空格符
    $code .= "\$s .= ' ' x ($coll + $numChars - length(\$s));";
    $code .= "    if (length(\$s) < ($coll+$numChars));";
    $code .= "\$tmp = substr(\$s, $coll, $numChars);";
    $code .= '$tmp .= "x (' . $numChars . ' - length($tmp));';
    $code .= "\$col .= '$delimiter' . \$tmp; ";
}

#-----
sub get_next_line {
    my($buf);

NEXTFILE:
    if ($open_new_file) {
        $file = shift @files || exit(0);
        open (F, $file) || die "$@ \n";
        $open_new_file = 0;
    }
    if ($size) {
        read(F, $buf, $size);
    } else {
        $buf = <F>;
    }
    if (! $buf) {
        close(F);
        $open_new_file = 1;
        goto NEXTFILE;
    }
    chomp($buf);
    # 将制表符转换成空格符 (假定制表符宽度为 8)

    # 首先扩展领头的制表符 —— 一般情况
    $buf =~ s/^(\\t+)/' ' x (8 * length($1))/e;

```

```

# 现在检索嵌套制表符。必须每次展开一个 —— 因此要使用 while 循环
# 在每次循环中，一个制表符被替换成余下的到下一个制表符为止的空格符
# 当不再有制表符时循环退出
1 while ($buf =~ s/\t/' ' x (8 - length($`)%8)/e);

$buf;
}

```

get\_next\_line 使用替换操作符的 /e 选项来去掉制表符。你能够猜出我们为什么要使用 while 循环而不是 /g 选项吗？原因在于，要想将制表符展为正确数量的空格符，我们必须知道这个和下一个制表符所在的确切位置。这也就意味着我们必须知道从行开始到制表符的字符数目，这可以通过计算 length(\$`) 得知。在下次循环中，这个长度还要考虑上一次展开的制表符。/g 选项虽然进行全局替换，但是它不再访问一个已经访问过的部分（也就是说，它总是一直向前操作）。如果使用这个选项，你将不可能知道任意时刻经过部分替换的字符串的长度。因此我们选择使用 while 循环来遍历包含制表符的字符串。

## 在超时中应用 Eval

每当你调用 eval 时，Perl 都将记下一旦在其中发生 die 调用时要执行的下一条语句。在内部，die 其实是引发了一个 longjmp 调用，因此 Perl 将控制转向 eval 后面的语句时，不管此时调用栈有多深都不花费任何时间。

在内部使用 setjmp 和 longjmp 提供了一种新的技巧：那就是终止阻塞的系统调用和无限循环。比如说，你希望在用户敲入些什么之前至多等待 10 秒（注 4）。如果你使用 \$buf=<> 的话，那么程序将被挂起，直到用户最终按下回车键为止，但是我们需要的是 Perl 在 10 秒钟后中止这项操作。产生一个超时并不困难；我们可以使用内建函数 alarm() 来在给定的若干秒钟后产生 ALRM 信号，如下所示：

```

$SIG{ALRM}=\&timed_out;
alarm(10);    # 告诉操作系统在 10 秒钟后发出 ALRM 信号
$buf=<>;      # 进入阻塞读状态

```

注 4： 感谢 Tom Christiansen 的这个例子。

无论当时 Perl 在执行什么，不管是在阻塞读状态还是死循环，`timed_out` 过程都将被调用（在 10 秒钟后）。问题在于 `timed_out` 如何让 Perl 放弃当时所正在执行的工作？这就是利用 `eval/die` 的好地方。你使用 `eval` 将 `$buf=<>` 括起来并在 `timed_out` 中放入 `die` 操作，于是控制将会传递到 `eval` 后面的语句中（这里是 `if` 语句），如下所示：

```
$SIG{ALRM}=\&timed_out;
eval {
    alarm (10);
    $buf=<>;
    alarm(0);          # 如果用户响应了输入操作就取消超时设置
};
if ($@ =~ /GOT TIRED OF WAITING/){
    print "Timed out.Processing with default\n";
    .....
}

sub timed_out{
    die "GOT TIRED OF WAITING";
}
```

如果用户在 10 秒钟之内没有敲击回车，`timed_out` 将被信号处理程序调用，继而调用 `die`，而它在内部将 `longjmp` 到紧跟最内层 `eval` 的语句中。当用户真的在设定的时间内敲击了回车时，`alarm(0)` 操作将重置 `alarm`。

注意，如果超时报警发生了，`$@` 中将包含下面的此类信息：“GOT TIRED OF WAITING at foo.pl line 100 (foo.pl 第 100 行等待超时)”。因此你不能使用 `eq` 而必须使用一个正则表达式（或是 `index` 操作符）。

Tom Christiansen 指出了一种微妙而有趣的情况。你必须在 `eval` 块中设置 `alarm`，这一点很关键。否则，在一台负载很重的机器上（而且对于小的超时值来说），有可能在调用 `alarm` 后错过下一个时间片，在控制还没有机会进入保护区前，这时程序重新获得了时间片，但是存在一种可能就是超时已经过期了，于是程序就会退出。

## 其他语言中的 Eval

让我们来看一下别的语言都对运行时计算和例外处理提供什么样的支持。

### Tcl

Tcl 解释器沿袭了典型的 shell 语法：每一条语句都是后面跟有一组参数的命令。如果在编译时命令是已知的，它就会产生字节代码并在以后执行，但是如果它是非固定的，那么解释器就会等到运行时再编译执行那条语句。（Tcl 的早期版本总是将程序当作字符串来看待，并且每次运行时都要对语句进行语法分析，即便是处于一个循环中也是这样。在这本书出版时，Tcl 解释器才刚刚朝着字节代码解释器的方向迈出了几步。）Tcl 支持一种用户级的 eval 调用，由它递归调用语法分析器，对这种字符串形式的后面跟有一组参数的命令进行解释执行。

Tcl 提供了等价于 Perl 中的 die 与 eval 的 error 和 catch 语句来进行错误处理。

### Python

Python 的 eval 函数可以计算并执行一个字符串，但是这个字符串不能包含有换行符。exec 语句则允许有换行符，但是由于 Python 要依赖于前导空白字符而不是一种显式的代码块结构，因此你需要正确的处理动态创建的、要提交给 exec 语句的字符串中的空白符。这要比正确处理 Perl 中代码块的作用域更为烦心。

Python 与 Perl 一样要经过一个编译与执行的阶段，而且对于每个名为 *module.py* 的模块，它都会产生中间字节代码并保存在一个称做 *module.pyc* 的文件中。在下次该模块被使用时，将会自动地加载这个中间字节代码文件。考虑到在编写这本书时 Malcolm Beattie 的 Perl 编译器正处在 alpha 测试阶段，所以可望不久的将来在 Perl 中也能够看到这种特性。

与 Java 和 C++ 一样，Python 通过支持作为语言一部分的例外类的概念，来进行例外处理。你可以使用 raise 来产生例外并使用 try/except/finally 的语法来捕获它们。（try 与 except 等同于 eval BLOCK 的形式。关键词 finally 用以表示在



不存在其他可以捕获该例外的语句的情况下,将调用这个默认的except代码块。)我特别喜爱Python解释器及Python库的这种自始至终一致的利用该机制的方式。

## C/C++

在这里不存在动态计算,但是却有大量可以连接到你的C应用程序中的解释器有公共领域中的或商用的,可以通过它们来支持C或C++风格的解释性语言。你可以从下面地址的自由编译器列表中查找到C-Interp或Xcoral应用: <http://www.idiom.com/free-compilers>。

C中没有用于进行例外处理的关键词。C++中有着一种与Java语言相同的try/catch/throw语法结构。例外可以是用户自定义的对象,而且还可以拥有自己的行为方法及私有数据。

## Java

Java与Perl一样也要经历相同的两个阶段:(1)编译成一种中间字节代码的形式;(2)执行这段中间代码。然而它所不允许的是凭空产生和计算新的代码。这里并没有什么技术上不可行的因素,因为javac编译器本身就是用Java来编写的,而且应当有可能将其打包成一个功能库,而不是独立的程序,同时还不违反任何安全限制。

在错误处理方面,Java有一种与Perl中的eval BLOCK方式等价的try/catch语法结构,其中所有的代码在编译时都是已知的。Java中的例外是真正的一级对象,因此与在Perl中必须进行字符串相比较,你能更好地区分它们。与Perl的die类似,Java中使用throw关键词来产生一个用户定义的例外。

Java进行严格的类型检查,并要求一个函数要枚举其可能抛出的例外(这被认为是函数原型的一部分)。因此如果你调用了一个抛出例外的函数,Java要求你的函数要么不重新将其抛出,要么必须将那个例外作为函数原形的一部分进行声明,如果你想将其传递出去的话。这样一来,每当你看到一个函数,你就会知道要进行处理的确切的一组例外,这对于一个由团队来开发的大型应用来说非常重要。依据你所持的不同观点,Perl中则没有任何此类功能或者限制。

## 相关资源

1. More Programming Pearls. Jon Bentley. Addison-Wesley, 1990

尤其与本章相关的是第 9 个专栏 “Little Languages”。

2. 运行时代码生成 (Run-time code generation)。

由 Don Pardo 收集的相关 WWW 链接和文章，地址在：

<http://www.cs.washington.edu/homes/pardo/rtcg.d/index.html>。

3. Perlsec。

Perl 与安全有关的文档。



### 本章简介:

- 包的基本知识
- 包与文件
- 包的初始化与销毁
- 私有性
- 符号的导入
- 包嵌套
- 自动加载
- 存取符号表
- 语言对比

## 第六章 模块

生活就是与周围事物的一种斗争,为了保持  
自我的存在。观念就是我们形成的  
战略计划,对攻击作出响应。

——Jose Ortega y Gasset 《国民的反叛》

`awk` 和各类 Unix shell 语言不能用于构建即便是并不怎么复杂的系统,其中主要原因就是它们缺少对模块化编程的支持。那里不存在拿起来就可以插入到系统中使用的代码实体;你不得不在其他独立的脚本中进行剪切和粘贴。这与诸如 Perl 这样的语言形成了鲜明的对比,它有着大量第三方模块 (module, 即功能库) 可以使用,这也是为什么 Perl 如此成功的原因。当进行语言之间的比较时,我认为可用的功能库要比纯语言特性重要的多。

Perl 允许你将代码划分成一个或多个可重用的模块。在这一章,我们将学习如何去完成下面的工作:

- 使用关键词 `package` 来定义模块。
- 使用 `use` 和 `require` 来加载预定义模块;我们在前面章节中已经看到了几个使用 `use` 的例子。
- 使用 “`::`” 记号来存取包的特定变量和子例程。
- 在运行时加载函数。

## 包的基本知识

关键词 `package` 标志着一个新的名字空间的开始。在它之后声明的所有的全局标识符（包括变量名，子例程，文件句柄，打印格式和目录句柄）都将属于这个包（`package`），例如：

```
package BankAccount
$total = 0;
sub deposit{
    my($amount)=@_;
    $total+=$amount;
    print "You now have $total dollars \n";
}
sub withdraw{
    my ($amount)=@_;
    $total-=$amount;
    $total=0; if $total <0;
    print "You now have $total dollars \n";
}
```

用户定义的全局标识符 `$total`、`deposit` 和 `withdraw` 属于包 `BankAccount`。包的作用域一直延伸到最内层的封闭代码块的结束（如果它在那个代码块中定义的话）或持续到另一个 `package` 语句的开始。在缺少显式的包声明的情况下，Perl 将假定此刻的包名为 `main`。

下面是你如何使用另一个包中全局符号的例子：

```
package ATM;                                # 开始一个新的名字空间
BankAccount::deposit(10);                   # 调用一个外部子例程
print $BankAccount::total;                  # 存取一个外部变量
```

要想存取位于另一个名字空间中的标识符，你就需要在变量名前添加包名；这被称为全能限定这个名字。注意，你必须使用 `$BankAccount::total`，而不是 `BankAccount::$total`；`$` 符后跟的是全能限定名。如果一个标识符名没有被全能限定，那么 Perl 将会在当前的活动包中进行查找。

既然 `package` 语句只表明一个有效的名字空间，因此你可以在不同的名字空间中自由切换：

```
package A;
$a=10;                # 这个 $a 位于包 A 中
package B;
$a=20;                # 这个 $a 位于包 B 中，而且同其他的 $a 完全独立

package A;
print $a;              # 打印结果为 10
```

C++ 程序员将会看出它们同 C++ 语言名字空间机制的相似性。

## 包与变量

在第三章“Typeglob 与符号表”中，我曾提到所有的全局名字都位于一个符号表中。这有点像一个善意的谎言。实际上每个包都有它自己的符号表，它们之间互不相同。（我们将在本章题为“存取符号表”一节中更多的谈到这个问题。）在包 main 中定义的标识符并不会被特别对待。除了有一点例外，那就是你还可以以另一种形式“`$::x`”来指代那个包中的变量 `$x`。

那些诸如 `$|`、`$_`、`@ARGV` 和 `%ENV` 之类的内建变量总是属于包 main 的。而且 Perl 允许你在其他包中直接使用它们，而无须加上前缀 `main::`。这些是 Perl 中唯一真正的全局变量。

你也许还记得词法变量 (`my`) 同符号和 typeglob 没有联系，因此也就与包没有任何关系。下面的表达将导致编译错误：

```
my $bankAccount::total;  # 错
```

这也就意味着你可以同时拥有两个类型和名字都相同的变量，只要一个为包全局变量而另一个为词法变量。下面这段代码是合法的，但是我们坚决不建议这么来写：

```
$x=10;                # 包 main 的全局变量
my $x=20;              # 文件范围的词法变量
print $x;              # 将打印出 20，词法变量的优先级更高一些
```

## 符号引用

我们前面已经见过符号引用的工作情况，它对变量和函数都有效。考虑下面的例子：

```
package A;
$x=10;

package B;
# 通过符号的方式来存取 $A::x
print ${"A::x"};

# 或以一种更为间接的方式
$pkg="A";
$var_name="X";
print ${"${pkg}::$var_name"};

# 间接的调用一个子例程
&{"A::foo"}(10,20); # 等价于 A::foo(10,20);
```

我们将在第八章“面向对象：下面的几步”中大量使用这种机制。

## 包与文件

同一个包的声明可以放在多个文件中。或者多个包可以在一个文件中进行声明。按照惯例，一个包通常有它自己的文件，而且其文件名为 *package.pm* 或 *package.pl*。以 *.pm* 为后缀的文件被称做 Perl 模块而在后缀为 *.pl* 的文件中的包通常则被称做库。现在推荐使用前面那种命名的方式，因为 *use* 语句要求这样，这一点我们马上就会看到。

关键词 *require* 只是把一个文件加载到程序中（在 shell 中的说法就是 *source* 它），这实质上与 C 语言中的 *#include* 相同，只是 Perl 用不着关心该文件是否已经装载过了（注 1）：

```
require "test.pl"; # 如果文件 test.pl 没有加载的话就加载它
```

注 1：与 C 或 C++ 的另一个重要区别就是，模块不用分成分离的声明和实现文件（头文件与“.c”文件），而且无须使用连接器将模块连接在一起。

如果你省略了后缀和引号，那么就假定使用 *.pm* 为后缀。use 语句在功能上与之类似只是更严格些，它只接受模块名而不是文件名。因此，尽管模块名与文件名总的来说并没有什么必然的联系，但 use 语句的确迫使你采用一种标准的命名习惯，在我看来这是很好的一件事。但是 use 的用途不仅限于这些句法上的要求。

use 与 require 之间最大的不同，就是 use 语句一旦被进行语法分析时就会执行。因此下面试图加载模块的代码并不能工作，因为赋值语句要在一切均被分析和编译后才能执行。

```
$pkg_name="Account";           # 在运行时执行
use $pkg_name;                  # 在编译时执行
```

实际上，这将会导致语法错；在这种情况下你必须使用 require。使用 use 的好处就在于一旦程序开始执行，就可以保证所有模块均被成功加载，也就不会在运行时出任何意外了。

use 与 require 的另一个重要差别将在后面的“符号的导入”一节中予以描述。

当使用 use 或 require 加载文件时，就会返回一个表示成功与否的布尔值(零表示失败，非零表示成功)。也就是说，在全局作用域中执行的最后一条语句必须是“return 1;”这样的，或者即便是“1”也行。注意，这并不一定得是文件中的最后一条语句；它只要是最后一条执行的语句即可。

## 加载路径

Perl 首先在内建数组 @INC 中指定的路径中查找使用 use 或 require 给定的文件。默认情况下，@INC 中包含了在解释器安装建立时指定的一些标准路径名。在我的机器上，@INC 的内容如下：

```
% perl -e 'print "@INC \n";'
/opt/lib/perl5/sun4-solaris/5.004 /opt/lib/perl5 /opt/lib/perl5/
site_perl/sun4-solaris
/opt/lib/perl5/site_perl/ .
```

你也可以使用 perl -V 来获得此类以及其他的配置信息。

你如果想要指定自己额外的目录，则可以使用下列做法：

1. 使用像 C 语言中那样的 -I 选项：

```
%perl -I/home/sriram/perl -I/local/mylib script.pl
```

我有时将自己使用和开发的各种版本的模块放在另外的目录中。有了这个选项就可以不用改动代码也能很容易的使用到这些模块。

2. 把环境变量 PERL5LIB 设置为由冒号分隔的路径信息。
3. 在调用 require 之前更改 @INC:

```
unshift (@INC, "/use/perl/include");          # 增加一条路径名
require 'foo.pl';
```

## 包的初始化与销毁

有时你需要在任何代码执行前来做一些初始化工作。Perl 的功能更为强大：它允许你仍处在编译阶段时就有机会执行代码。

一般情况下，Perl 在对文件进行语法分析时会编译整个文件，而当编译成功完成后，从第一条全局语句往后开始执行。然而，如果它在语法分析时碰到了名为 BEGIN 的子例程或代码块，它不仅将其编译，而且还会立刻执行，接着才继续完成对剩余文件的编译工作。为说明这一点，我们做一个小实验：

```
sub BEGIN {          # 还可以表示为 BEGIN { }; "sub" 一词是可选的
    print "Washington was here \n";
}
foo***;              # 故意设置的错误
```

上面的代码将打印出如下信息：

```
Washington was here
Syntax error at x.pl line 4, near "*** ;"
Execution of x.pl aborted due to compilation errors.
```

虽然带有语法错误的程序一般根本就不会执行，然而位于错误之前的 BEGIN 子例程还是得到了执行。



因为甚至在编译阶段完成之前，BEGIN代码块就得到了执行，所以它可以影响未完成的编译过程。如果你想手工编写程序中的包含路径的话，下面就是一种方法：

```
BEGIN {  
    Unshift (@INC, ".../include");  
}  
use Foo;                                # 首先在 ".../include" 中查找 Foo.pm
```

一种更简单的方式就是使用 Perl 发行版中所带的 lib 模块：

```
use lib qw(.../include);                # 将路径添加到 @INC 前面
```

就像你想要在其他任何代码执行前，要做初始化工作一样，你有时也希望在所有代码执行完毕时做一些收尾工作。当程序将要退出时，无论它是成功退出还是由于别的原因退出，END代码块都将被调用。也就是说，即便程序是因为诸如算术例外等原因退出，END块也会被执行。如果程序退出是因为未能捕获的信号所引起的，那么该代码块将不被执行。

BEGIN和END来源于*awk*。而且Perl还支持多条BEGIN和END语句。BEGIN语句根据其出现的先后次序来执行，而END语句则以出现的先后顺序的相反顺序来执行（后进先出）。如果存在带有许多BEGIN和END代码块的多个包，那么还应考虑包加载的先后顺序。

## 私有性

Perl中的符号是可以任意存取的；信息隐藏并不是强制性的。联机文档中就形象的描述道：“Perl并不强制进行模块中私有和公有部件的划分，而这可能是你在其他语言如C++，Ada或Modula-17中已经习惯的特性。Perl并不热衷于强制的私有性。它更喜欢的方式就是，如果你没有被邀请进入它的客厅，那么就呆在外边，但这不是说它带着猎枪（注2）。”

---

注2： 我有一次在一段C++代码中看到了这个宝贝：“#define private public”，它就在要包含的头文件的前面。如此强烈需要数据的人一定能想办法得到的。

除了可以存取外部包的现存变量或子例程,一个包还可以在另一个包的名字空间中创建新名字,这一点我们以前就见到过。考虑下面的代码:

```
package Test;
# 在另一个包中创建一个变量和一个子例程
$main::foo=10;
sub main::myFunc {
    print "Hello \n";
}

package main;
myFunc();                # 将打印出 "Hello"
```

尽管在一般的应用中这么来写并不怎么恰当,但是如果控制的好的话,该特性可以有很好的用途。你可以用它来把外部包的符号名导入到你自己的名字空间中。我们将在下一节来研究这个问题。

## 强制私有性

你可以在文件范围内使用 `my` 操作符来获得绝对安全的私有变量。因为它们和包没有任何联系,所以无法从别的域中进行存取(在这里就是指文件范围内)。但是由于它们与包没有任何关系,它们将至多被限制在文件范围以内。考虑下面的例子:

```
package A;
my $a=10;                # 一个词法变量

package B;
print $A::a;              # 包 A 中没有这个变量
print $a;                 # 将打印出 10, 这是因为它可以看到词法变量
                          # (即便是包 B 有效)
```

如果你想创建私有的子例程名该怎么办呢? 你无法使用 `my` 来声明一个子例程名,但是你可以使用匿名子例程并将它们的引用放置到词法变量中:

```
my $rs_func = sub {
    ....
};
```

现在,可以在所在域中对 `$rs_func` 进行间接访问(如果它是位于文件中任意位置的全局变量),但是你无法在别的文件中存取到它。在调用这个子例程时,你可以使用 `&$rs_func`, 或者如果你要大量的调用该子例程的话,还可以使用 `typeglob` 别名,这样会方便和高效一些:

```
{
    local (*func)=$rs_func;
    for (1..100){func()};
}
```

尽管可以隐藏你自己的全局标识符,但你却无法阻止别的模块向你的名字空间中安装新的名字。实际上,老一些的 Perl 库大量的利用了这种自由的特性。(例如,你可以看一下 Perl 标准库中的包 *bigint.pl*。)

## 符号的导入

有时为了省去输入的麻烦,你可能需要有选择的将一些符号导入到自己的名字空间中。例如,你想以 `sqrt` 来代替 `math::sqrt` 或者以 `deposit` 代替 `BankAccount::deposit`。 `use` 语句可以让你指定一组可选的用来导入的函数名:

```
use BankAccount('withdraw','deposit');
withdraw();                                # 现在你无须使用全能限定就能调用它了
```

就模块来说,它必须已经准备好向任何 `use` 它的程序输出这些名字(也只有这些名字)。而且它还应在用户未指定名字列表时有相应的对策。这两项工作都交由称做 `Exporter` 的标准模块来处理。`BankAccount` 包可以按照如下所示来实现:

```
package BankAccount;
use Exporter;
@ISA=('Exporter');          # 继承 Exporter
@EXPORT_OK=('withdraw','deposit');

sub deposit { .... }
sub withdraw { .... }
```

这段代码先加载 `Exporter` 模块,然后再使用数组 `@ISA` 继承它。现在暂且认为这样能够工作;我们很快就要学到继承的问题了。数组 `@EXPORT_OK` 用来说明要输

出哪些符号。模块的使用者于是就可以使用 `use` 语句来指定一个包含在 `@EXPORT_OK` 中定义的一个或多个符号的列表。如果用户这么写：

```
use BankAccount('deposit');
```

那么与 `withdraw` 不同，`deposit` 函数无须使用全能限定就能够被调用。要想告诉 `Exporter` 模块不在你的名字空间中输出任何符号，只需保持列表为空即可。

如果模块使用 `@EXPORT` 来代替 `@EXPORT_OK` 的话，那么用户将输出缺省的所有符号，而不管在输入列表中是否提及这些符号。我建议，作为一名模块编写人员，还是使用更礼貌的 `@EXPORT_OK` 好一些。

请查看 `Exporter` 的文档来了解其他众多的功能。其中的一项允许模块的使用者使用标识名（tag name）来成组的导入函数，或者使用正则表达式来指定相应的组。

## use 和 Exporter 是如何工作的

如果你对 `use` 和 `Exporter` 是如何工作的细节不感兴趣的话，你可以简单的跳过本节而不会损失连贯性。这一节包含了一种“为了学习知识而学习的知识”的内容。

下面的语句：

```
use BankAccount ('withdraw', 'deposit');
```

等同于：

```
BEGIN { require BankAccount ;  
        BankAccount::import('withdraw','deposit');}
```

`BEGIN` 保证了这条语句在进行语法分析时就会被执行。如果 `BankAccount.pm` 还未加载的话，`require` 将负责加载，`import` 调用的是那个模块上的子例程（注3）。

注3： 这是一种善意的谎言。它实际上完成的是 `BankAccount->import`（使用箭头而不是`::`），一种稍有不同的调用子例程的方式。我们将在第七章“面向对象编程”中具体学习这个记号。现在暂且这样的解释就足够了。

Import 并不是 Perl 的一个关键词，它只是对一个名为 import 的用户定义模块的调用，模块可以以任何形式来定义它，并以任何需要的方法来操纵其参数。如果 BankAccount 没有定义 import 和继承它的话，那么 use BankAccount 与 require BankAccount 之间就没有任何区别。通过使用 Exporter，用户就可以继承一个 import 方法而无须自己实现它。

为了理解 Exporter 是如何工作的，让我们自己来创建一个 import 子例程。我们将开发一个简单的称做 Environment 的模块，它可以使我们快速的存取环境变量。下面是如何来使用这个模块：

```
use Environment;
print $USER, $PATH;
```

我们现在可以不必使用 \$ENV{'USER'}，只是简单的写作 \$USER 即可。换句话说，模块 Environment（更确切的说是模块中名为 import 的函数）在调用者的名字空间中安装了诸如 \$USER 和 \$PATH 的变量。

例 6-1: Environment.pm: 创建与环境变量对应的变量

```
package Environment;
sub import {
    # 获取调用者的详细信息：包名，当前文件名，和行号
    my ($caller_package)=caller;
    foreach $envt_var_name(keys %ENV){
        *{"${caller_package}::$envt_var_name"}=\$ENV{$envt_var_name};
    }
}
1; # 用来表示初始化成功
```

为了使该例子更小一些，import 忽略了它的参数表。它使用调用者的内建函数来获得调用包的名字并在那个包中创建别名。对于环境变量 USER 来说，foreach 中的语句可以翻译成下面的形式：

```
*{"main::USER"}=\$ENV{USER};
```

我们假定调用者为包 main。

这一小段代码覆盖了第三章的大部分知识点。表达式的右部返回一个指向包含环

境变量值的标量变量的引用,并赋值给一个`typeglob`。(还记得我们讨论过的有选择的别名吗?)左边的`typeglob`表达式在`main`的符号表中创建了一个新的条目,而它的标量变量部分被用来指向从右部得出的值。`Exporter`的工作方式与此类似,只不过它只创建函数的别名。

巧的是,标准 Perl 发行版中包含了一个名为 `Env` 的模块,其功能与我们的包 `Environment` 非常相似。唯一的不同就是 `Env` 不是创建环境变量的别名,而是使用 `tie` 机制(注4),这是一种比上面讲的更为低效的实现方式。

## 包的嵌套

既然所有包在作用域上都是全局性的,因此并不支持包嵌套。然而你却可以有两个包,一个称做 `A`,另一个称做 `A::B`,这给出了一种包嵌套的错觉。这只是一种命名习惯,并不表示这两个包之间有任何关系;但是这种习惯常常应用在之间互有关系的包之间,在这种情况下,嵌套包这个字眼似乎并不错。例如,你可以有一个用来操纵矩阵的称做 `Math::Matrix` 的包,和另一个名为 `Math::Poisson` 的用来模仿队列模型的基础支撑。它们之间唯一的关系就是它们本质上都与数学相关;但是它们没有共享任何实现特性。

`::`记号仍像以前一样用来存取嵌套包的变量和子例程:

```
$p=Math::Poisson::calculate_probability($lambda, $t);  
print $Math::Constants::PI;
```

当你使用 `use File` 时,你会记得 Perl 将会查找名为 `File.pm` 的文件。当你使用 `use Math::Poisson` 时, Perl 将查找名为 `Math/Poisson.pm` (`Math` 为目录, `Poisson.pm` 为文件)的文件。`::`记号被翻译成了一个路径分隔符,这是因为冒号对于 DOS 来说有特殊的意义。Perl 在嵌套层次上不加限制。

---

注4: 我们将在第九章“绑定”讨论绑定方案。

## 自动加载

比如说，如果你调用了一个名为 `Test::func()` 的函数，而 `func()` 并没有在 `Test` 模块中定义，Perl 则会自动寻找一个称做 `Test::AUTOLOAD()` 的子例程。如果这样的子例程存在，Perl 将会以刚才传递给 `func()` 的参数来调用它。同时，名为 `$AUTOLOAD` 的变量将被设置为刚才所调用函数的全名（“`Test::func`”）。Objective-C 程序员会发现这同声明 “:forward” 有点相似，在那里一个对象使用这条语句来捕获所有它不能处理的过程调用，以便将它转交给一个“代理”。

`AUTOLOAD` 子例程可以做它任何想做的事。例如，它可以完成下面其中的一种工作：

- 由它自己处理调用。`Test::func` 的调用者并不知道实际上是 `AUTOLOAD` 处理了这个调用。
- 自动的凭空创建一个子例程（使用 `eval`）来完成相应的工作。实际上，你可以不用调用那个子例程，而只需跳转到那里执行即可，如：

```
sub AUTOLOAD {  
    .....创建子例程.....  
    goto &$AUTOLOAD;          # 跳转到那里  
}
```

这是一种特殊形式的 `goto` 语句，它将删除 `AUTOLOAD` 子例程的堆栈帧以至于 `Test::func` 不知道它是在 `AUTOLOAD` 中被调用的。

- 使用标准的 `Dynaloader` 模块来动态加载一个对象文件（在 Microsoft Windows 中叫做 DLL），然后执行相应的调用。这也是 `AUTOLOAD` 的一种最常用的方式。
- 使用 `system` 函数来执行另一个拥有相同文件名的程序。Perl 库中的 `Shell.pm` 模块就是用来完成这种工作的一个奇妙的版本。这里有一个简化的版本：

```
#-----  
package Shell;  
#-----  
sub AUTOLOAD {  
    my ($program) = $AUTOLOAD;
```

```

    # 我们只是对命令名而不是包名感兴趣
    $program=~s/^.*/:/:;
    system("$program @_");
}
#-----
use Shell;
ls('-lR');          # 因为 ls() 并不存在, 所以将触发一个对 AUTOLOAD 的调用
mail('-s "This is a test" joe@foo.com <letter.txt');

```

自动加载还可以被用来延迟对子例程的加载, 直到绝对必需时才进行加载。有一个名为 `Autosplit` 的模块, 它被用来将一个模块分成多个模块, 每个模块中包含来自于原始模块中的一个子例程, 这样就可以使用 `Autoloader` 模块来加载与当前调用子例程对应的文件。

## 存取符号表

Perl 有许多的功能都支持内省 (introspection), 其主要能力就是获取符号表内容的信息。这个属性有时也被称做反射 (reflection)。

反射可以使我们方便的编写诸如调试器 (debugger) 和剖析器 (profiler) 等系统工具。我们还将第十一章“对象持续性的实现”中, 使用这个属性来开发一种能够无须编写任何应用相关代码, 就能透明的将对象数据卸载到文件或数据库中 (并在以后重新提取出来) 的模块。

在本章的前面我们已经看到了, 每个包都有它自己的符号表 (也称做 *stash*, “symbol table hash” 的缩写)。Perl 将这些符号表制成可用的普通散列表。一个名为 `Foo` 的包的符号表可以通过称做 `%Foo::` 的散列表来进行存取。包 `main` 的符号表可以通过 `%main::` 或其简单形式 `%::` 来存取。实际上所有其他的包的散列表都可以通过包 `main` 的散列表 (因此 `%main::` 指向它自己) 来存取, 如图 6-1 所示。

对一个包中的所有符号名进行迭代很简单:

```

foreach $same (keys %main::){
    print "$name, \n";
}

```



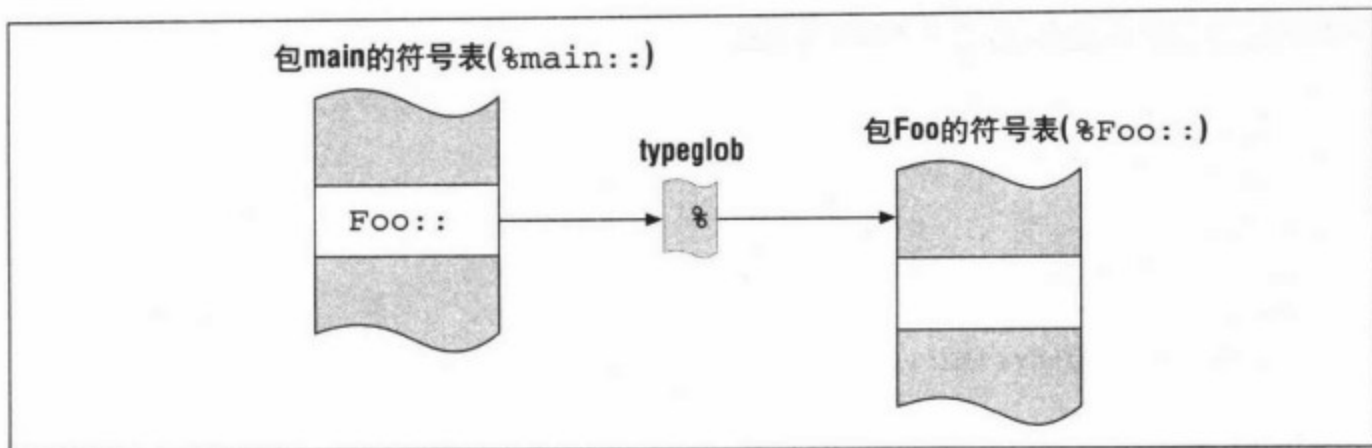


图 6-1 包的符号表在包 main 的名字空间中得到

正如我们前面所见到的，每个这种符号名都映射到一个 typeglob，而它又指向一个或多个值（每种类型有一个或更多：标量变量、数组、散列表、子例程、文件句柄、打印格式名或目录句柄）。不幸的是，没有直接的办法来检查某个值是否真的存在。例 6-2 描述了一种将一个给定包中的所有变量打印出来的方法，同时也演示了一种对于给定的 typeglob 找出哪个值已经存在的办法。

例 6-2: 打印出包中的所有符号

```

package DUMPVAR;
sub dumpvar {
    my ($packageName)=@_;
    local(*alais);                                # 一个局部 typeglob
    # 我们希望根据特定的包名来存取相应的 stash
    *stash = *{"$packageName":};                  # 现在 %stash 就是符号表
    $,="";                                          # 打印分隔符
    # 对符号表中的项进行迭代，符号表中包含以符号的名字为键值的 glob 值
    while (($varName, $globValue)=each %stash){
        print "$varName=====\n";
        *alias=$globValue;
        if(defined($alias)){
            print "\t \$$varName $alias \n";
        }
        if(defined(@alias)){
            print "\t \@varName @alias \n";
        }
        if(defined(%alias)){
            print "\t \%varName ", %alias, "\n";
        }
    }
}
  
```

下面的代码演示如何使用 DUMPVAR:

```
package XX;
$x=10;
@y=(1,3,4);
%z=(1,2,3,4,5,6);
$z=300;
DUMPVAR::dumpvar("XX");
```

打印结果为:

```
x=====
      $x=10
y=====
      @y 1 3 4
z=====
      $z 300
      %z 1 2 3 4 5 6
```

dumpvar 通过连续的为每个 typeglob 创建一个别名, 然后再枚举每个类型, 看它是否已经定义。需要认识到的重要一点是, 它只打印出了最顶层的全局数据, 因为通过各种引用来指向的匿名数据结构根本就没有被打印出来。

## 与其他语言的比较

在这一章中, 我们已经看到了 Perl 所提供的名字空间的分割, 有限的存取保护, 内省, 运行时子例程加载 (自动加载), 包的初始化与销毁结构以及将名字输出到不同的名字空间中等内容。让我们带着这些功能特性来看一看其他语言的情况。

### Tcl

Tcl 的“包”提供了一种基本的为一组代码标注一个版本号的特性, 以供代码的使用者显式的指明它所要求的版本。如果没有匹配的版本的话, Tcl 就会设置一个错误标志。Perl 也提供对版本号的支持 (我们将在下一章中更多的谈到这个问题)。

这里的包并不像 Perl 中的那样以全局名字空间的形式工作。相反, Tcl 支持一种多个解释器同处一个程序中的概念, 来提供各名字空间之间彻底的分立。这种机制被用于创建 SafeTcl, 由它来提供用以完成代码计算的安全及不安全的隔间。我们将在第二十章“Perl 的内部工作”中看到, Perl 的包 Safe 在内部使用了一种截然不同机制(既不是使用多个包也不是多个解释器)。

就内省而言, Tcl 提供了一个“info”命令来查找有关全局符号的内容。它没有继承特性, 但是却有几个诸如 `[incr Tcl]` 和 `stoop` 的自由软件扩展, 它们通过在基本的语言上建立一个面向对象层来弥补这一缺憾。

动态函数调用实现起来稀松平常; 你只需将命令名指定为一个变量, 接着在运行时它就能够被展开和执行了。

## Python

Python 提供了与 Perl 非常相似的包机制。每个 Python 模块都是一个名字空间(一个以名字为键值的词典或散列表), 而且 Python 还允许你对词典进行遍历和查询。与 Perl 类似, 它也不强制实施名字空间的私有性, 并交由程序员来判定是否遵循模块的边界。作为一个模块使用者, 你可以将一些特定的符号输入到自己的名字空间中。(这里没有等价于 `@EXPORT` 的机制, 而我认为这件好事。) Python 没有提供任何类似于 Perl 的文件范围的词法变量的工具, 这也就意味着如果需要的话, Python 却不能为你提供真正的私有性, 这一点与 Perl 是不一样的。

## C/C++

在所有这里提及的语言中, C 与 C++ 的动态性最差的; 它们着重在编译时将所有的内容都能够具体化, 这样代码在运行时就只需全速执行了。

C++ 中的虚函数提供了函数在运行时的联编。虽然编译时的类型检查确保了这种联编的安全, 与更为动态的诸如 Objective C 甚至 Java 等语言比较而言, 它的代码还是烦琐一些。

C++ 支持 RTTI (运行时类型识别), 但是这项功能仅限于发现指针的实际类型并对其进行动态转换。(如果指针的转换有误, 运行时环境就会抛出一个例外。) 这种机制并不能够告诉你一个变量所实际指向的内容。

## Java

Java 提供了两个级别上的模块化: 包与类, 这里包是一组类的集合。(我们将在下一章中学习类的概念。) Perl 的 `package` 等价于这两种情况。Java 不允许一个包去操纵另一个包的名字空间(即不能输出), 但却允许一个包有选择的输入它所需要的类。它将大量的精力投放在了安全上, 但实际上却没能阻止那些执着的黑客的攻击。Perl 则是通过第三方的名为 `Safe` 和 `Penguin` (它依赖于 `Safe`) 的包来试图获取与之相似的隔离特性(而且也不能保证安全)。

自从 Java Bean 和 JDK 1.1 出现以来, Java 获得了相当程度的自省能力, 尽管它还无法匹敌像 Perl 所提供的那样多的公共可用信息。就是否提供这些信息而言, 各有各合理的争辩理由; 所有这一切都将导向不同的编程模型。在 FORTRAN 和 COBOL 盛行的时候, 人类就被送上了月球, 这也证明了如果你不是把时间花在争论语言之间孰优孰劣上的话, 你就可以完成许多要做的事儿。

Java 允许你以字符串的形式指定函数名来动态的“分发”一个函数调用, 并在该函数不存在时捕获一个例外; 这一点有些类似于使用 Perl 的符号引用。



本章简介:

- 面向对象简介
- Perl 中的对象
- UNIVERSAL
- 习惯的更新
- 与其他面向对象语言的比较

## 第七章

# 面向对象编程

有个天天向前走的孩子，他只要观  
看什么东西，他就变成了它。

——惠特曼（译注1）

《有个天天向前走的孩子》

面向对象是一种最新的软件方法学，它充斥于媒体的宣传中，甚至到了凡是使用“面向对象设计”这个字眼就一定表示是一种好设计的程度。在这一章中，我们将学习那些所宣传的究竟都是些什么以及如何利用Perl来创建对象。我将使用大量有关面向对象的文献来使你相信：存在一种好的折衷方案，而且对象革命确实是一件好事。

如果你已经对面向对象有所了解，那么只需阅读一下附录二“语法总结”中的“对象”一节。那里还提供了一个将C++的例子转换成等价的Perl代码的范例。

## 面向对象简介

Fred Brooks 在他的经典著作《The Mythical Man-Month》讲到：

那些正为空间不够而苦恼的程序员，常常在抛开代码的束缚之后，回头来更多地思考自己的数据时，才能做得最好。表达是编程的本质。

他是在谈论减少空间的使用问题，尽管如此，这仍不失为智者之言。

---

译注1：惠特曼（1819 - 1892），美国著名诗人，《草叶集》的作者。

复杂的系统从根本上说是分层的,而且为了利用这一点,人们发明了大量的抽象概念和方法论。直至70年代末,功能分解(自顶向下的设计)一直是理解和实现复杂系统的最可靠的方法。一个开发人员一般从书写高层的伪代码开始,然后将每一部分进一步细化,直至它足够详尽而能够翻译成一种实现语言。Niklaus Wirth称这种方式为逐步求精。后来就出现了结构化的方法,SA/SD (Structured Analysis/Structured Design, 结构化分析/结构化设计) 占领了主导地位。这种方法使用了大量的工具和记法,如数据流图、过程规格、数据字典、状态转换图和实体关系图等来进行设计、编制文档和开发系统。重点仍旧放在系统开发的过程化方面,而不是动态(状态转换)或结构化(数据)的方面。

最近大约15年里主要的认识是,系统的功能与它所操纵的数据相比更易于变化。一个用于记录保存雇员详细信息的人事信息系统就需要了解那么多的信息。而另一方面它的功能却要包含记录管理层的改组、税法、医疗保险变更还有人力资源部主管热闹的上任和静悄悄的离职等内容。

这种新的认识,使得处理问题的方式完全颠倒过来。数据和数据的结构化方式现在是第一重要的,而且代码包裹着重要的数据被组织到模块中。其益处是巨大而且立竿见影的。

首先,数据库与代码是同步的,因为代码是根据数据来组织的。有人抱怨面向对象程序同关系型数据库“阻抗失配”,但是那是由于关系型数据库系统受限于简单的数据类型;关系与对象模型之间并不存在根本的不匹配。诸如Informix/Illustrator和Oracle已于近期开始在它们的关系数据库系统中提供对抽象数据类型的支持。

专注于数据还有另一项好处:数据结构一般是一些你能够联系起来的東西。例如,一个航空公司拥有飞机、航线和飞行区间等主要实体。在设计一个飞行规划系统时,这些实体能够提供很好的讨论、分析和设计的中心。任何要着手进行新的设计文档写作而又不知如何开始的人肯定会欣赏这种方式的! 最终的设计与实现同样更易于理解(也就会更容易维护),因为解释起来很简单。Fred Brooks在《The Mythical Man-Month》一书中谈到:“给我看你的流程图却把数据表藏起来,那么我仍然会搞不明白。让我看你的数据表,那么我一般不会需要你的流程图;因为它们很清晰。”

最后，一个被划分成以数据为中心模块的系统，可以简单的在一组程序员间进行分配。对给定数据或一组相关数据的变更只能由其“所有者”来完成；那个开发人员将成为项目中其他人员的部件提供者。

面向对象是沿着这条途径的最新进展。代码不仅是以数据为中心，它还尽力来封装（隐藏）实际的数据结构，而只对外界开放有限的、具备良好文档的接口：这是一组知道如何来操纵数据结构的函数。这些数据结构被称做对象（object）。录像机、手表、汽车和其他现实世界中的物体都是一些我们希望仿效的优秀实例，因为它们成功地在简单的接口后面隐藏了所有无数的复杂细节（当然，录像机上闪动的“12:00”表明仍然有不少接口需要简化）。虽然你肯定可以使用传统的诸如C或COBOL甚至汇编语言来实现封装良好的以数据为中心的设计，但是面向对象语言所提供的两种特性则不仅仅是出于句法上的方便：这就是多态性和继承。我们将会看到这些特性是如何方便地使我们得以来构建可重用的模块的。

需要强调的是，面向对象方法学与SA/SD在功能、动态和结构方面是相似的。但是它们在方式和侧重点上差别巨大；面向对象设计方法学先将注意力放在数据抽象上，最后才是过程的抽象。

## Perl 中的对象

让我们开始在 Perl 中实现对象之前，先来定义几个预备性的术语。

一个对象（还可以称作一个实例，instance）就像一辆给定的轿车，拥有下面的这些内容：

- 属性（attribute）又称特性（property）（如颜色：红色；座位数：4；动力：180 马力）
- 标识（identity）（如我的这与你的不同）
- 行为（behavior）（如它可以驾驶，向前和向后移动）

某个特定类型的对象被称作属于一个类（class）。我的车与你的车都属于叫做汽

车的类，或者如果你并不特别关心具体的类别，那么还可以属于名为车辆的类。一个类的所有对象都具有相同的功能。

在这一节，我们将学习如何创建对象，和如何通过使用继承和多态性来丰富我们的基本设计。

## 属性

一个对象就是一组属性的集合。像我们在第二章“实现复杂的数据结构”中所讨论的那样，我们可以用数组或散列表来表示这个集合。例如，如果你需要记录下雇员的详细情况的话，你可以选择下面的一种方式：

```
# 使用散列表来储存雇员的属性
%employee=("name"=> "John Doe";
           "age"=>    32,
           "position"=> "Software Engineer");
print "Name: ", $employee{name};

# 或者使用数组
$name_field=0;$age_field=1;$position_field=2;
@employee=("John Doe", 32, "Software Engineer");
print "Name: ", $employee[$name_field];
```

第八章“面向对象：下面的几步”中“高效的属性存储”一节中描述了一种更为有效的属性存储方案。同时我们将在我们所有的例子中使用散列表。

## 唯一标识

很明显，一个%employee是不够的。每个雇员都要求有一个唯一标识和他或她自己的属性集合。你可以动态的分配这个数据结构，也可以返回一个指向局部数据结构的引用，如下所示：

```
# 使用匿名散列表
sub new_employee{
    my($name,$age,$starting_position)=@_;
    my $r_employee={
        "name"      => $name,
        # 使用匿名散列表
        # 创建一个唯一的对象
```



```

        "age"      => $age,
        "position" => $starting_position
    };
    return $r_employee;          # 返回对象
}

# 或者返回指向局部变量的引用
sub new_employee {
    my ($name, $age, $starting_position) = @_;
    my %employee = (
        "name"      => $name,
        "age"       => $age,
        "position"  => $starting_position
    );
    return \%employee;          # 返回指向局部对象的引用
}

# 使用它来创建两个雇员
$emp1 = new_employee("John Doe", 32, "Software Engineer");
$emp2 = new_employee("Norma Jean", 25, "Vice President");

```

在上面的两种情况下, `new_employee()` 都将返回指向一个唯一数据结构的引用。

作为该子例程的使用者, 你不必知道这个标量变量包含的是对一个Perl数据结构的引用, 还是一个字符串(例如, 它可能仅仅包含一个数据库主关键字, 而剩余的详细信息在一个团体的数据库中)。于是雇员的信息的细节进行了很好的封装。、  
要注意, 不要把封装同强制私有性搞混了。

在上面的例子中, 散列表就是对象, 而指向散列表的引用则称之为对象的引用。要知道自从上一章到现在, 我们还没有引入任何新的语法。

## 行为

所有存取或更新对象一条或多条属性的函数, 共同组成了对象的行为。考虑下面的例子:

```

sub promote_employee {
    my $r_employee = shift;
    $r_employee->{"position"} =
        lookup_next_position($r_employee->{"position"});
}

```

```
}

# 使用方法
promote_employee($emp1);
```

这样的函数在面向对象的环境中也被称作实例方法 (instance method)，因为它需要一个类的特定实例，在上面的这种情况下是指一个雇员。

为了避免必须为每个方法添加后缀 "\_employee"，我们将把所有的这些函数放入它们自己的包中，取名为 Employee：

```
package Employee;
sub new{      # 不再需要后缀
    ....
}
sub promote{
    ....
}
```

为了使用这个模块，你可以这么写：

```
$emp=Employee::new("John Doe", 32, "Software Engineer");
Employee::promote($emp);
```

你会发现，这段代码已经开始封装了一个称之为 Employee 的类：这段代码的使用者调用的只是接口函数 (interface function) new 和 promote，而且并不知道或关心用于存储雇员详细信息的数据结构类型，或像我们前面提到的那样，是否在内部使用数据库。

## 多态性的必要性

我们现在顶多看到了一个 C 程序员所能够完成的工作，除了他或她可能希望使用一个 struct 结构来保存属性以外。这恰好是 stdio 库完成工作的方式，比如，fopen() 是一个返回指向动态分配的唯一 FILE 结构的指针的构造函数。这个指针（对象的引用）被提交给其他的诸如 fgets() 和 fprintf() 的方法。

不幸的是，当问题更加复杂时麻烦也就会出来了。假设我们要保留有关小时工和

正式雇员的信息。小时工按小时数来拿工资而且有资格享受加班费，而正式工则每月发一次薪水。一种实现方式就是为每种类型的雇员创建一个 new 函数：

```
package Employee;
# 创建正式雇员
sub new_regular {
    my ($name, $age, $starting_position, $monthly_salary)=@_;
    my $employee = {
        "name"      =>$name,
        "age"       =>$age,
        "position"  =>$starting_position,
        "monthly_salary" =>$monthly_salary,
    };
    return $employee; # 返回对象的引用
}
# 小时雇员
sub new_hourly {
    my ($name, $age, $starting_position,
        $hourly_rate, $overtime_rate)=@_;
    my $employee={
        "name"      =>$name,
        "age"       =>$age,
        "position"  =>$starting_position,
        "hourly_rate" =>$hourly_rate,
        "overtime_rate" =>$overtime_rate
    };
    return $employee; # 返回对象的引用
}
```

现在如果我们想得到一个雇员一年到现在的工资情况，我们就必须区分这两种类型的雇员。我们可以提供两个子例程，`compute_hourly_ytd_income()`和`compute_regular_ytd_income()`，但是这样并不算完。小时工与正式工在其他方的不同（如可用假期、医疗补助等等）或者再引入其他类型的雇员（如临时工）都将导致函数的组合爆炸。更糟糕的是，接口会要求包的使用者先区分雇员类型之后才能调用恰当的函数。

为了使我们走出这个误区，我们将不同类型的雇员放在不同的包中。然后我们使用关键词`bleed`在内部将对象标记为指向它们所在包的指针。下面例子中黑体显示的行为我们对上面代码所做改动的行（解释跟在后面）：

```

#-----
package RegularEmployee;
sub new {
    my ($name, $age, $starting_position, $monthly_salary)=@_;
    my $r_employee = {
        "name"      =>$name,
        "age"       =>$age,
        "position"   =>$starting_position,
        "monthly_salary" =>$monthly_salary,
        "months_worked" =>0,
    };
    bless $r_employee, 'RegularEmployee';      # 在对象上标注包名
    return $employee;      # 返回对象的引用
}
sub promote{
    #...
}
sub compute_ytd_income{
    my $r_emp=shift;
    # 假定 months_worked 属性在某个时刻被修改过
    return $r_emp->{'monthly_salary'}*$r_emp->{'months_works'};
}

#-----
package HourlyEmployee;
sub new {
    my($name, $age, $starting_position,
        $hourly_rate, $overtime_rate)=@_;
    my $r_employee={
        "name"      =>$name,
        "age"       =>$age,
        "position"   =>$starting_position,
        "hourly_rate" =>$hourly_rate,
        "overtime_rate" =>$overtime_rate
    };
    bless $r_employee, 'HourlyEmployee';
    return $r_employee;
}
sub promote{
    #...
}
sub compute_ytd_income {
    my ($r_emp)=$_[0];
    return $r_emp->{'hourly_rate'}*$r_emp->{'hours_worked'}
}

```

```
    + $r_emp->{'overtime_rate'}*$r_emp->{'overtime_hours_worked'};  
}
```

bless 以一个普通的指向数据结构的引用为参数。它将会把那个数据结构（注意，不是引用本身）（注1）标记为属于某个特定的包，于是这样就赋予了它更强大的功能，这一点我们很快就会看到。bless 对我们的散列表来说就好比给孩子的洗礼。它并不对数据结构做任何改动，就像洗礼除了给他们一个附加的身份之外不会改变一个人一样。

bless 的好处在于它提供给我们一种直接使用该对象的方式，如：

```
# 首先像以前一样创建两个对象  
$emp1=RegularEmployee::new('John Doe', 32, # 多态性  
                             'Software Engineer', 5000);  
$emp2=HourlyEmployee::new('Jane Smith', 35, # 多态性  
                            'Auditor', 65, 90);
```

现在我们使用箭头记号来直接调用实例的方法，或者用面向对象的话说，调用对象的方法：

```
# 直接调用  
$emp1->promote();  
$emp2->compute_ytd_income();
```

当 Perl 看到 `$emp1->promote()` 时，它会决定 `$emp1` 属于哪个类（也就是在其中执行 bless 的）。在这里，它是 `RegularEmployee`。Perl 于是就会如下所示调用这个函数：`RegularEmployee::promote($emp1)`。换句话说，箭头左边的对象只是作为相应子例程的第一个参数。

与在 C++ 中不同，记号 `::` 和 `->` 实际上都是允许的。Perl 的实例方法没有什么神奇的地方。它只是第一个参数碰巧为对象引用的普通子例程。（你也许已经注意到了 `promote` 方法从前面一节开始就没有改变过。）

那么这只是句法上的考虑吗？最终我们似乎所得到的，就是以另一种记号来调用对象实例方法的能力。

---

注1： 引用就像 C 语言中的 `void *`。对象是有类型的，而 C 指针或 Perl 引用则没有。

不，我们获得的是另一项重要的好处。那就是模块使用者无须再使用if语句来区分不同类型的对象，而是让Perl来将调用导向相应的函数。也就是说，不用再这么写了：

```
if(ref($emp) eq "HourlyEmployee"){
    $income=HourlyEmployee::compute_ytd_income($emp);
}else{
    $income=RegularEmployee::compute_ytd_income($emp);
}
```

我们可以简单的写成：

```
$income=$emp->compute_ytd_income();
```

Perl 这种调用相应模块函数的能力被称做运行时联编 (run-time binding)。顺便提一下，大家还记得第一章“数据引用与匿名存储”中，ref函数返回一个表示引用所指向实体类型的字符串，而对于经过bless的对象引用，它返回相应类的名字。

注意，在处理工资记录时，\$emp可以在一次迭代中为正式雇员，而另一次则可以是小时雇员。这种特性被称做多态性 (polymorphism) (一个对象可以有多种形式的功能)。

多态性与动态联编是面向对象语言的主要贡献。它们使系统拥有巨大的灵活性，因为你现在无须改变工资处理代码就可以添加一个新类型的雇员。(而同其他类型的雇员拥有相同的接口。) 这种情况之所以可行，就是因为每个对象“知道”如何来计算自己到目前的收入情况。下面这句最重要的规则值得记住：

如果你发现自己在使用条件语句来区分对象类型，那么就表明程序设计缺乏灵活性。

这种设计之所以具有灵活性的原因还在于，你可以为任意的包添加新的方法，而不会对已存在的代码造成任何影响。

## 类方法和属性

类属性是指那些属于一个类中所有对象的属性,而不会只在单个雇员上发生变化。例如,一个保险公司会为全体雇员提供健康保证金,因此将该公司的名字存入每个雇员的记录中没有什么意义。

类方法(也称做静态方法)是指那些无须特定的对象实例就能够工作的从属于类的函数。例如,一个名为 `get_employee_names()` 的子例程,无需一个雇员对象就能够知道自己应该怎么做。

与C++或Java不同,Perl中没有针对类属性和方法的特定的语法。类属性只是包的全局变量,而类方法则是不依赖任何特定实例的普通子例程。Perl对这些普通的子例程就支持多态性和运行时联编(而不仅仅是实例方法),这可以被用来实现真正灵活的设计。

考虑下面的例子:

```
$details=<STDIN>;          # 输入由制表符分隔的雇员详细信息
($type, $name, $age, $position)=split(/\t/, $details);

# 创建相应类的雇员对象
$emp = $type->new($name,$age,$position);

# 现在就可以像往常一样来使用这个对象了
$emp->compute_ytd_income();
```

在这个例子中, `$type` 中可能包含这两种字符串中的一种: “HourlyEmployee” 或 “RegularEmployee”。注意这个变量不是对象,它只是类的名字。我们通过避免将包名字进行硬性编码的方法,改进了前面一节的这个例子。这为什么称得上是种改进呢? 哦,如果没有这种机制,你就必须使用类似下面的方法来创建相应类型的对象:

```
if ($type eq "HourlyEmployee"){
    $emp = HourlyEmployee->new(...);
}else {
    $emp = RegularEmployee->new(...);
}
```

任何明显的要依赖对对象类或类型进行检查的代码，都需要太多的维护。如果你明天要增加一种新类型的雇员，那么你就必须回去把这种新类型加入到所有这些代码中。

回想一下实例方法的情况，箭头左边的对象将作为该子例程的第一个参数传递。这里也一样，过程HourlyEmployee::new同样需要重写成下面这种所期望的形式：

```
package HourlyEmployee;
sub new {
    my ($pkg, $name, $age, $starting_position, $hourly_rate,
        $overtime_rate)=@_;
```

因为实例和类方法都是普通的子例程，因此你总是可以通过检查所传递的第一个参数的类型来编写一个子例程使它具有它们任何一个的功能。考虑下面的构造函数，它根据调用方式的不同，可以创建一个新对象或克隆一个已经存在的对象：

```
package Employee;
sub new {
    $arg =shift;
    if(ref($arg)){
        #以 $emp->new()的方式调用：克隆给定的 Employee 对象
        #....
    }else{
        #以 Employee->new()的方式调用：创建一个新的 employee 对象
        #....
    }
}
```

你现在可以这样使用这个方法：

```
# 把new()用作类方法
$emp1=Employee->new("John Doe", 20, "Vice President");

# 把new()用作一个实例的方法来克隆雇员信息
$emp2=$emp1->new();
```

我将让你来回答为什么你会需要克隆雇员对象！

我们在这一节学到了什么呢？如果我们在编写类方法时，都默认以模块的名字为



它的第一个参数的话,我们就可以使模块的使用者能够应用运行时联编和多态性。我们从现在起就按照这个方法走。

你或许会对类方法为什么需要提供它自己的模块名感到好奇。等我们很快讲到继承时,就可以来回答这个问题了。

### 迂回策略: 一种间接的记号

Perl 如果不能以多种方式来满足每人不同的喜好的话,它就不是 Perl 了。它支持除箭头记号的另一种方式,称做间接记号,这是一种将函数名放在对象或类名的前面的方法。让我们举一个例子就会更清楚些:

```
$emp = new Employee ("John Doe", 20, "Vice President");
```

C++ 的程序员就会认出这种记号。这种方法同样可以应用到对象中:

```
Promote $emp "Chairman", 100000;           # 给他升职加薪
```

注意到在 \$emp 和它的第一个参数 ("Chairman") 之间没有逗号。Perl 就是通过这种方式来知道你是在使用间接记号来调用一个方法而不是当前包中的一个子例程。你也许会在下面的例子中发现更多的东西:

```
use FileHandle;
$fh=new FileHandle(">foo.txt");
print $fh "foo bar\n";
```

这里 print 是 FileHandle 模块的一个方法。

虽然这种间接记号同箭头记号拥有相同的效果,但是它不能在级联调用中使用:

```
use FileHandle;
$fh=FileHandle->new(">foo.txt")->autoflush(1);  # 级联调用
```

### 继承的必要性

Perl 允许一个模块在一个特殊的名为 @ISA 的数组中指定一组其他模块的名称。当在模块中找不到某个类或实例方法时,它就会检查那个模块的 @ISA 是否被初

始化, 如果已经初始化了, 它就会查看其中的某个模块是否支持这个“缺少”的函数, 它会调用发现的第一个函数并将控制转交给它。这个特性被称做继承 (heritance)。考虑下面的代码:

```
Package Man;
@ISA=qw(Mammal Social_Animal);
```

它指定了 Man (人) 是一种 Mammal (哺乳动物), 而且还是一种 Social\_Animal (社会型哺乳动物)。所有哺乳动物常见的特征 (方法) 都将在 Mammal 类中得到支持, 而无须再在 Man 类中实现。让我们来看一个更实用的例子。

在我们试图区分小时雇员同正式雇员的时候, 我们却走向了另一个极端, 使他们完全相互独立。很明显, 他们之间有许多作为雇员所共同拥有的属性 (姓名, 年龄和职位) 和行为 (如, 提升)。我们于是就可以使用继承将这些共同的方面“提取”出来, 作为一个名为 Employee 的超类 (superclass) (或称基类, base class):

```
#-----
package Employee;                                # 基类
#-----
sub allocate{
    my ($pkg, $name, $age, $starting_position)=@_;
    my $r_employee=bless {
        "name"      =>$name,
        "age"       =>$age,
        "position"  =>$starting_position
    }, $pkg;
    return $r_employee;
}
sub promote{
    my $r_employee =shift;
    my $current_position = $r_employee->{"position"};
    my $next_position    =lookup_next_position($current_position);
    $r_employee->{"position"}=$next_position;
}
#-----
package HourlyEmployee;
#-----
@ISA= ("Employee");                               # 从 Employee 继承
sub new{
    my($pkg, $name, $age, $starting_position,
        $hourly_rate, $overtime_rate) = @_;
```

```

# 让 Employee 包来创建和 bless 对象
my $r_employee=$pkg->allocate($name, $age,
                               $starting_position);

# 最后增加 HourlyEmployee 所特有的属性
$r_employee->{"hourly_rate"}      =$hourly_rate;
$r_employee->{"overtime_rate"}    =$overtime_rate;
return $r_employee;              # 返回对象的引用
}

sub compute_ytd_income{
    ....
}

# ... 包 RegularEmployee 与之类似

```

所有雇员共有的特性都在基类中实现。既然 HourlyEmployee 和 RegularEmployee 都需要一个类方法 new() 来分配散列表，然后进行 bless，接着将共有的属性插入到这个表中，我们因此将这个功能分离出来放在模块 Employee 中的一个可继承的子例程 allocate 中。

注意 allocate 是如何避免将类名硬编码到其中的，这样一来就可以保证最大限度的重用性。HourlyEmployee::new() 调用 \$pkg->allocate 意味着 allocate 的第一个参数，\$pkg 拥有值 HourlyEmployee。allocate 使用这种方式而将对象直接 bless 到被继承的类中。HourlyEmployee::new 无须再创建对象，它只需将自己特定的属性插入其中。

从用户的角度来看没有发生任何变化，你仍然可以这么写：

```
$emp=HourlyEmployee->new(....);
```

但是我们现在已经设法去除了模块中的冗余代码，并将其开放以供将来的提高。

## 重载基类

假定我们需要确保小时雇员永远不应该升至经理的职位。下面的例子将告诉你，如何通过重载 (override) 基类中的 promote 方法来进行这项检查：

```

package HourlyEmployee;
sub promote{
    my $obj=shift;

```

```

    die "Hourly Employee cannot be promote beyond 'Manager'
        if ($obj->{position} eq 'Manager');
    # 调用基类的 promote
    $obj->Employee::promote();      # 要显式指定包的名字
}

```

这段语法告诉 Perl 开始在 @ISA 的层次结构中从类 Employee 开始来查找 promote()。这使我们很难改变对继承层次的看法。为了避免这种情况的发生, Perl 像 Smalltalk 一样提供了一个称做 SUPER 的伪类 (pseudo class), 因此你可以这样来写:

\$obj->SUPER::promote(); 它会搜索 @ISA 的层次结构来寻找合适的 promote 子例程。现在如果我们在 Employee 和 HourlyEmployee 之间的继承关系中再放入另一个包的话, 我们只需更新 HourlyEmployee 的 @ISA 数组。

---

**注意:** 我们现在已经逐步摆脱在调用一个模块的子例程时使用 :: 记号。一个子例程要么直接导入你的名字空间中, 在这种情况下你无须使用全能限定名, 或者使用 -> 记号来调用。你在存取外部包的变量时仍然需要使用 "::" 记号。

---

## 对象的销毁

Perl 自动垃圾收集引用记数为零的数据结构。如果一个数据结构被 bless 进入了一个模块, Perl 允许那个模块在对象销毁之前做一些收尾工作, 这样需要调用那个模块中的名为 DESTROY 的特殊子例程, 并传递给它要销毁对象的引用:

```

Package Employee;
Sub DESTROY{
    My ($emp)=@_;
    Print "Alas, ", $emp->{"name"}, "is now no longer with us \n";
}

```

这同 C++ 语言中的析构函数, 或 Java 中的 finalize() 有些类似。Perl 进行自动的内存管理, 但是那个对象被释放以前你还有机会来做一些事情。(与 Java 的 finalize 不同, Perl 的垃圾收集是确定的; 只要已经不再有任何东西指向这个对象, DESTROY 就会被马上调用。)

注意，你并不是必须要声明这个子例程，你只有在有一些收尾工作需要完成时才这么做。在像 Socket 这样的模块中，你会需要关闭相应的连接，但是对于像 Employee 这样并不使用任何外部系统资源的对象来说，你不必提供一个 DESTROY 方法。回想一下当一个函数没有被找到时将会调用 AUTOLOAD。在你提供 AUTOLOAD 但没有提供 DESTROY 方法的情况下，你也许希望确保 AUTOLOAD 来完成这种可能性的检查：

```
sub AUTOLOAD{
    my $obj = $_[0];
    # $AUTOLOAD 中包含有所缺方法的名字

    # 永远不要传播 DESTROY 方法
    return if $AUTOLOAD =~ /::DESTROY$/;
    # ....
}
```

## 存取方法

Rumbaugh 等人所写的《Object-Oriented Modeling and Design》一书中讲到：

当一个类的代码可以直接访问另一个类的属性时，封装就遭到了破坏。直接存取就会对数据的存储格式和地点做出假定。这些细节必须被隐藏在类中..... 存取另一个对象的属性的恰当的方式，就是通过调用对象的操作来“去要”，而不是简单的“去拿”。

这对于以继承关联起来的类和互不相关的类同样适用。

为了防止直接对对象的属性进行存取，我们可以提供“存取方法”。下面的这两个方法将会读区取和更新一个雇员的“position”属性：

```
$pos=$emp->get_position();           # 读属性
$emp->set_position("Software Engineer"); # 写属性
```

更为常用的方式是通过一个方法来同时处理读和写：

```
$pos=$emp->position();                # 读属性
$emp->position("Software Engineer"); # 写属性
```

下面是该模块可能的实现方式:

```
package Employee;
sub position{
    my $obj=shift;
    @_? $obj->{position}=shift      # 更改属性
      : $obj->{position};           # 读取属性
}
```

注意在这两种情况下（读和设置）该方法均将返回最后的 position 的值，这是因为表达式 \$obj->{position} 是最后一个计算的。

每次你要存取属性时都要调用一个方法，这似乎简直是在浪费时间。但是，事实证明，存取方法对于要考虑变化的设计来说是绝对必须的。我们来考虑以下的好处：

### 封装 (Encapsulation)

存取方法将会隐藏对象属性是如何存储的。如果你改变了数据的存储方式，只有这些方法才需要改动；而剩下的代码，包括导出类都可以保持不变。Perl 像其他面向对象的脚本语言一样，为了提高性能和空间效率有时会需要进行重新设计，这时存取方法就是一个好东西。在象 Smalltalk、CORBA (Common Object Request Broker Architecture, 公共对象请求代理体系) 和 ActiveX 等其他著名的技术中，存取属性的唯一方式就是使用存取函数。

### 副作用 (Side effect) (译注 2)

存取方法有时被用来在获取或更新属性时触发执行某些操作。GUI 工具箱就例行地利用这种专业用语。比如：

```
$button->foreground_color('yellow');
```

上面的代码不仅更改了前景颜色属性的值，同时还更新了屏幕。

### 存取检查 (Access checking)

存取方法可以被用来禁止更新。例如，诸如雇员姓名的关键字属性应当一旦建立就不能更新；存取方法可以轻而易举的施加这种控制。

---

译注 2： 请注意，不是副作用。

### 计算属性 (*Computed attribute*)

一个雇员的收入可以被视为是一个属性, 尽管它在内部需要计算得知。我们不是编写一个 `compute_ytd_income()` 这样的方法, 而是简单的称它为 `income()`。这可以使它看起来像一个属性存取方法, 而且还可以阻止对这项属性的更新。

**体会:** 要养成编写存取方法的习惯。在下一章我们将会学到一个称作 `ObjectTemplate` 的模块, 一个名为 `Class::Template` 的标准库, 和 CPAN 上的一个叫做 `MethodMaker` 的模块, 它们可以为你自动生成存取方法, 因此实在没有不去使用这些方法的理由。

**警告:** 即便你的属性包裹在存取方法之中, 你也要警惕使用这些方法的毫不相关的类。当回顾一段代码时, 一定要查看这些存取的实际目的; 有时通过提供其他的方法来去掉这种不必要的存取, 或许更好一些。例如, 一个用户应当总是使用 `$emp->promote()`, 而不是直接的去更新 `position` 的属性。

## UNIVERSAL

所有模块都隐含的继承了一个称做 `UNIVERSAL` 的内建模块, 并继承了如下三个方法:

### `isa` (包名)

例如, 如果 `Rectangle` 模块继承了 (无论是以间接的方法) `Shape` 模块, `Rectangle->isa('Shape')` 将返回 `true`。

### `can` (函数名)

如果 `Rectangle` 或它的任何基类包含有名为 `draw` 的函数, `Rectangle-can('draw')` 将返回 `true`。

### `VERSION` (版本号)

如果你这样写:

```
package Bank;  
$VERSION = 5.1;
```

并且模块的用户这样写:

```
use Bank 5.2;
```

Perl 将会自动的调用 `Bank->VERSION(5.2)`, 而它, 举例来说, 则可以确保所有 5.2 版所要求的库均被加载。由 `UNIVERSAL` 提供的默认 `VERSION` 方法, 在 `Bank` 的 `$VERSION` 变量值小于模块用户所需要的值时, 将简单的退出。

因为 Perl 允许一个包任意的修改其他的名字空间, 于是一些包就通过使用 `UNIVERSAL` 模块来为它们希望输出给每一个人的子例程提供一块保留区域。我建议你自已不要使用这个“功能”(或者至少不要在贡献给 CPAN 的代码中使用)。

## 方法的查找

我们已经提到, 当 Perl 在目标模块中找不到一个方法时, 它会搜索的两个地方: 继承层次 (`@ISA`) 和 `AUTOLOAD`。在检查继承层次时, Perl 同样要检查基类的 `@ISA` 数组, 并采取一种深度优先的搜索策略, 将使用第一个找到的方法。让我们来仔细查看一下对所有这些子例程确切的搜索顺序。例如,

```
package Man;  
@ISA=qw(Mammal Social_Animal);
```

对 `Man->schmooze` 的调用将产生下面的搜索顺序, 首先检查的是继承层次:

1. `Man::schmooze`
2. `Mammal::schmooze`
3. (`Mammal` 的基类, 递归的)::`schmooze`
4. `Social_Animal::schmooze`
5. (`Social_Animal` 的基类, 递归的)::`schmooze`
6. `UNIVERSAL::schmooze` (因为 `UNIVERSAL` 被隐含的放在每个模块 `@ISA` 数组的末尾)

接着 `AUTOLOAD` 也以同样的顺序进行查找:

7. `Man::AUTOLOAD`



8. `Mammal::AUTOLOAD`
9. `(Mammal 的基类, 递归的)::AUTOLOAD`
10. `Social_Animal::AUTOLOAD`
11. `(Social_Animal 的基类, 递归的)::AUTOLOAD`
12. `UNIVERSAL::AUTOLOAD`

控制将会交给第一个可用的子例程并且搜索终止。如果所有的搜索都失败了, 那么 Perl 将产生一个运行时例外。

## 习惯的更新

虽然 Perl 在如何组织模块上提供了无比的灵活性, 我们还是选择坚持本章介绍的一组特定的规则, 这样每个人都可以以一致的风格来使用模块。让我们来快速总结一下这些惯例:

- 模块必须有属于自己的文件 `<module>.pm`。(要记住最后一句执行的全局性语句必须返回 1 以表明成功的加载。)
- 模块中的所有子例程都应当被设计成方法。也就是说, 它们的第一个参数应当是类名或对象引用。为了更加方便, 它们应当处理任何一种情况。
- 包名应当永远不会硬编码到代码中。你必须总是使用从提供给  `bless`  的第一个参数中获得的包名。这样可以使构造函数能够被继承。
- 总是为类和实例的属性提供存取函数。

下面的例子将练习使用所有的这些技巧和惯例。

## 例子

考虑一个销售电脑和配件的商店。每个配件都有型号、价格及折扣等内容。顾客可以购买单独的配件, 但是也可以使用特定的配件来组装一台定制的电脑。商店

将为最终的价格加收销售税。这个例子的目的是向你提供这所商店的每一件商品的净价。

我们需要考虑到这样的事实，那就是一件商品还可能由其他的商品组成，销售税有赖于部件的类型和消费者所在的地区，还有就是我们可以向组装电脑收取组装费。

开始一项设计的一个有用的技巧，就是使用 Ivar Jacobson 在《Object-Oriented Software Engineering》一书中所提出的“使用黑箱分析”的方法。你站在用户的观点来看待应用的接口，而不用担心特定对象的属性。那样一来，我们就可以理解对象的接口而无需担心其内部实现细节。让我们来看一下我们是如何来使用这个系统的：

```
$cdrom =new CDRom ("Toshiba 5602");

$monitor =new Monitor("Viewsonic 15GS");
print $monitor->net_price();

$computer =new Computer($monitor, $cdrom);
print $computer->net_price();
```

图 7-1 显示了一种设计对象模型的方法。我使用 Rumbaugh 的 OMT (Object Modeling Technique, 对象建模技术) 记号来描绘类、继承层次和类之间的联系。那个三角表示“是”一种关系，而带有 1+ 的行表示一对多的关系。计算机“是”一件商店里的商品，它包含其他部件（“有一”关系）。一个 CD-ROM 或显示器是一个部件，而且还是一件商品。

所有店内商品所共有的属性被放置在 StoreItem 类中。为了计算任何商品的净价，我们必须最后要考虑折扣和销售税；我们不能将所有部件的净价简单的进行相加。因此，我们将计算分成两步：price，它会从价格中减去折扣部分，以及 net\_price，它要加上销售税。到目前为止，component 类均为空类，因为所有的功能都放置在 StoreItem 中。很明显，如果问题到此为止，那么这种设计就没必要这么复杂；我们只需设计一个包含价格和折扣的查找表和一个用于计算价格的函数就行了。但是我们这里是为考虑将来的变化而设计。我们希望，在我们开始考虑不同地点的税值，处理包含其他部件的部件和收取劳动费时，它能够逐渐丰满起来。最好是一开始就采用一种整体上的思路。

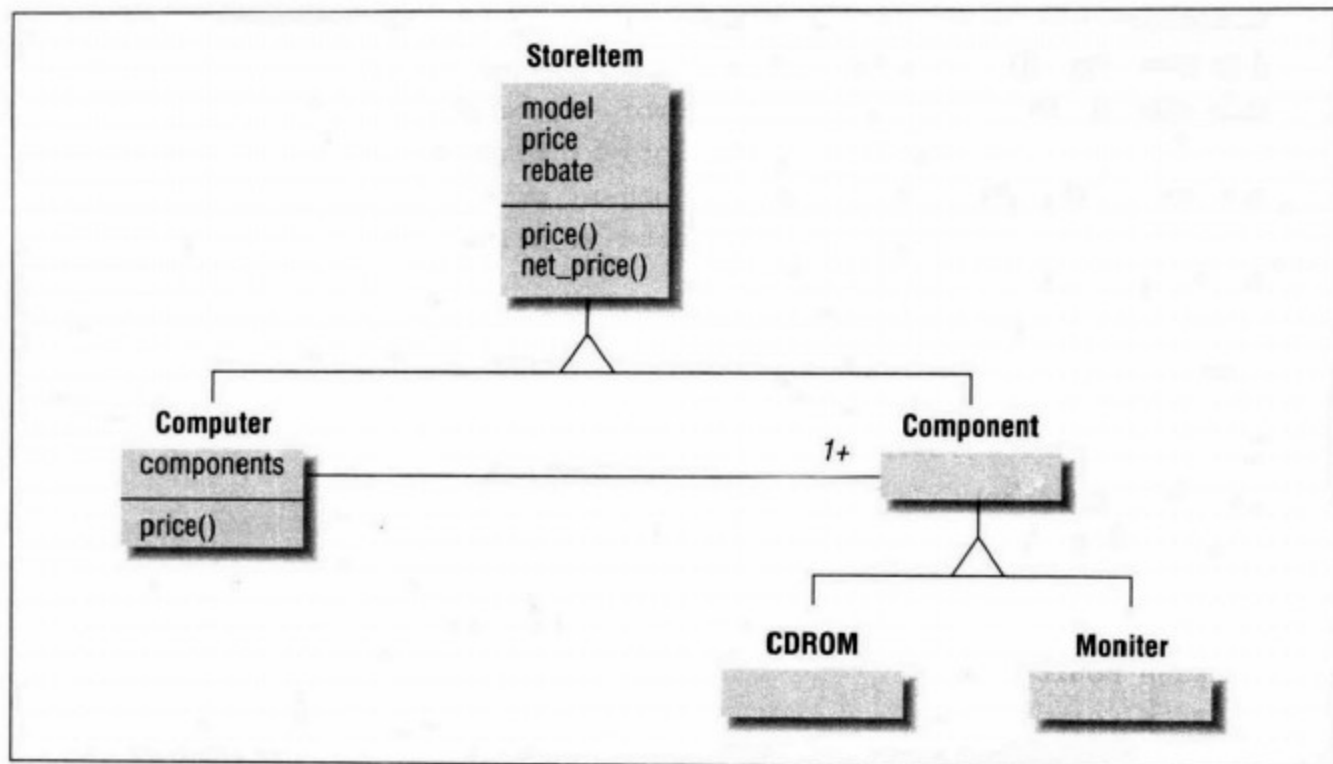


图 7-1 计算机商店例子的对象模型

Computer 类并不使用它的 price 属性；相反，它累加其组成部分的价格。它不必重载 net\_price 的功能，因为那个函数并不考虑对象的类型，只是为对象的价格加上销售税。

例 7-1 将对象模型翻译成代码。

例 7-1：对象实现范例

```

package StoreItem;

my $_sales_tax = 8.5;      # 将所有部件的折扣后价格加上 8.5%

sub new {
    my ($pkg, $name, $price, $rebate) = @_;
    bless {
        # 所有属性名前都被加上一个下划线，用以表明它们是私有的
        # (这只是一种惯例)
        _name => $name, _price => $price, _rebate => $rebate
    }, $pkg;
}

# 存取方法
  
```

```

sub sales_tax {shift; @_ ? $_sales_tax=shift:$sales_tax}
sub name {my $obj=shift; @_ ? $obj->{_name}=shift:$obj->{_name}}
sub rebate {my $obj =shift; @_ ? $obj ->{_rebate}=shift
            : $obj->{_rebate}}
sub price {my $obj =shift; @_ ? $obj->{_price}=shift
            : $obj->{_price}-$obj->{_rebate}}

sub net_price {
  my $obj=shift;
  return $obj->price * (1+$obj->sales_tax/100);
}
#-----
package Component;
@ISA=qw(StoreItem);

#-----
package Monitor;
@ISA=qw(Component);
# 现在临时将价格和折扣硬性编码到代码中
sub new { $pkg=shift; $pkg->SUPER::new("Monitor", 400, 15)}

#-----
package CDROM;
@ISA=qw(Component);
sub new { $pkg=shift; $pkg->SUPER::new("CDROM",200,5)}

#-----
package Computer;
@ISA = qw(StoreItem);

sub new {
  my $pkg=shift;
  my $obj=$pkg->SUPER::new("Computer", 0, 0); # 设置价格哑元
  $obj->{_components}=[];                    # 部件列表
  $obj->{components(@_)};
  $obj;
}

# 存取方法
sub components{
  my $obj=shift;
  @_ ? push (@{$obj->{_components}}, @_): @{$obj->{_components}};
}
sub price {
  my $obj=shift;

```

```
my $price=0;
my $component;
foreach $component ($obj->components()){
    $price+=$component->price();
}
$price;
}
```

这个例子就体现了为了变化而设计的哲学。所有的实例变量都有存取方法，这样也就使我们有可能在 `Computer` 类中重载 `price()`。存取方法 `Computer::components` 可以在以后加以改动，以检查不同部件之间的兼容性。甚至连包的全局变量 `$sales_tax` 也是通过存取方法来存取的，因为我们期望今后不同的部件将会征收不同的销售税，因此我们从对象来获取销售税值。

同样要注意的是，构造函数使用 `SUPER` 来存取其超类的 `new` 程序。以这种方式，如果你明天创建一个 `Component::new` 方法，其余任何包都不需要改动。`StoreItem::new` 将把对象 `bless` 到传递给它的包中。它没有将包名硬编码到程序中。

如果你将这些包放到不同的文件中，回想一下第六章“模块”，那么这些文件应该符合 `<包名>.pm` 的命名惯例。而且，它们应当以 `“1;”` 或者 `“return 1;”` 作为最后一条执行的语句。

## 与其他面向对象语言的对比

### Tcl

Tcl 的基础库没有任何面向对象的功能。它最近拥有了一种包结构，能够为子例程和全局变量提供名字空间（包之间不存在任何关系）。Tcl 是一种颇具延展性的语言，因此有好几个可以自由使用的库来尽力为该语言添加一种面向对象的结构。一种名为 `stoop` 的包就提供了一套纯 Tcl 的解决方案，功能包括单一或多重继承，动态联编，运行时类型识别，等等。另一个叫作 `[intr Tcl]`，这是一种颇具抱负的成果，它提供一组类似于 C++ 的关键词和机制，尽管它还需要给 Tcl 打个补丁程序。

## Python

Python 是一种用以学习面向对象的优秀语言。(它碰巧也是我最喜爱的面向对象脚本语言。)所有的机制,包括诸如链表和散列表的内部数据结构和外部功能库都拥有一致的面向对象界面。Python 提供了许多让类开发人员来编写不同类型存取方法的钩子,并支持多重继承。与在 Perl 中不同,你必须挑选出一种表达形式(更乐观的来看待这个问题,就是你有选择最佳表达形式的自由),Python 中的所有对象都是通过散列表来实现的。

## C++ 和 Java

在 Perl 和 C++ 面向对象的实现上存在许多极为不同的地方。

- **对象结构。**与 Perl 不同, C++ 要求你使用关键词 `class` 来定义对象的结构。而 Perl 中则不关心你是如何保存对象的状态的——以散列表、数组还是标量变量的形式。Perl 唯一要求的是你返回一个经过 `bless` 的指向那个数据的引用。
- **私有性。**C++ 有多种关键词来保证不同程度的私有性(`private`、`protected`、`public`)。Perl 并不强制实施私有性。如果你需要私有性的话,则可以使用词法变量。
- **构造函数/析构函数。**C++ 要求对象的构造子例程拥有与类相同的名字。Perl 则没有任何此类结构——任何子例程都可以成为构造函数(使用名称 `new` 只是个习惯问题)。在另一方面,在对象将要被销毁时,Perl 和 C++ 都需要众所周知的析构函数名。C++ 的构造函数实际上就是一个初始化者;在构造函数获得控制之前,内存就已经分配了。在 Perl 中分配和初始化均是程序员的责任。
- **静态方法与实例方法。**C++ 中提供 `static` 关键词来区分静态函数和实例方法。Perl 则不做区分——子例程之间是没有区别的。Perl 的子例程可以通过检查它的参数来成为任何一种形式。
- **声明与定义。**和 Perl 不同, C++ 要求类的声明独立于它的实现(除非是内嵌的实现方式)。典型的 C++ 惯例是将声明放在一个头文件中,而实现放在另外的文件中。

- **编译时功能与运行时功能。**C++ 要求所有的类信息，如继承层次、属性和方法的数量及类型等等要在编译时知道。Perl 则允许在运行时重新定义任何东西；你可以增加、删除或更新方法，或者通过改变 @ISA 来改变继承层次。我建议你不要利用这种能力。
- **运行时联编。**由于C++进行严格的类型检查，因此运行时联编只在对象继承自一个公共基类时才可用。而 Perl 则没有这种限制。

上面比较中谈到的大多数 C++ 的情况，对于 Java 来说也是适用的。

## 相关资源

1. perltoot (Perl 文档). Tom Christiansen。

“Tom的面向对象指导教程”精辟的讲解了面向对象尤其是Perl中的面向对象技术。必读。

2. comp.object。

FAQ 地址: <ftp://rtfm.mit.edu/pub/usenet/comp.object>。是已编纂的最好的FAQ之一。

3. Object-Oriented Modeling and Design. J Rumbaugh, M.Blaha, W.Premarlani, F.Eddy 和 W. Lorensen. Prentice-Hall,1991。

精辟的讲解面向对象，尤其在转换成程序时。还包含了面向对象与其他软件方法的比较。

4. Design Patterns: Elements of Reusable Object-Oriented Software. Erich Gamma, RichardHelm, Ralph Johnson 和 John Vlissides. Addison-Wesley,1994。

该书是一部有关常用对象交互模式的分类讲解（它是与语言无关的）。即使这些模式本身有时是显而易见的，但是给它们起个名字也会丰富从事对象研究人员的词汇表。

5. Bring Design to Software. Terry Winograd. Addison-Wesley, 1996。

这本书尤其探究了几个成功的软件产品,并得出面向用户的设计是最好的软件设计方法的结论。(实际销售的产品中没有一个特别在意面向对象编程的。)有趣而有说服力。

6. The Mythical Man-Month. Frederick P. Brooks. Addison-Wesley, 1995。
7. Object-Oriented Software Engineering: A Use Case Driven Approach. Ivar Jacobson. Addison-Wesley, 1994。





本章简介:

- 高效的属性存储
- 代理
- 关于继承

## 第八章

# 面向对象： 下面的几步

没有任何纽带会像继承这样联系得如此紧密。

—— Stephen Jay Gould

本章主要讲述的是一组有关 Perl 对象的思想、方法和建议。我并没有将这些内容混杂在一起来讲。下面是要讲述的主题：

### 高效的属性存储

我们将不使用散列表，而是寻找一种另外的途径来表示对象属性。本章考查的两种策略将占用更少的空间而且速度更快。

### 代理

如何使用 AUTOLOAD 来转发方法调用。

### 继承与组合

讲述我所认为的继承的坏处以及结构化类的其他方法。

## 高效的属性存储

我们一向使用散列表来存储对象属性。我们这样做有几个好的理由：

- 每个属性都是自我描述的（也就是说，每个属性的名字和类型可以很容易的从对象中获得），这样可以很容易的来编写可读性好的代码。它还可以使模块无需对象显式合作就能够进行自动的对象持续性存储或对象的可视化。

- 位于继承层次中的每个类都可以自由独立的添加属性。
- 实际上，每个实例（而不仅仅是类）都可以拥有唯一的一组属性而且还可以在运行时加以改动。人工智能领域的人们经常使用这种基于 *slot* 或 *frame* 的方式，因为它能够很好的调整自己以适应新的信息。

当然，并不是所有的问题都需要这种程度的一般性。而且，虽然 Perl 的散列表速度快（是数组速度的 15% 以内），相当的精简（键值字符串不会进行复制），但它们并不是完全的低开销。创建 100 个对象就意味着你将拥有 100 个散列表，而每个散列表都要为将来优化插入操作而分配额外的存储空间。

这一节描绘了另外两种解决方案，一种是使用数组而另一种则是使用 `typeglob`。两种方法的普遍性比起使用散列表都差一些，但是速度更快而且更节省空间。第一种方法是为本书开发的名为 `ObjectTemplate` 的模块（注 1）。另一种利用了 `typeglob`，而且在一些标准的 CPAN 模块中也存在有限的应用，最著名的有 `IO` 和 `Net` 模块。我对推荐这种方法存有疑虑，因为它太有点“另类”了。但是我在这里提出来可以使你更好的理解这些标准模块。

## ObjectTemplate:使用数组来存储属性

这一节展示的模块使用数组来存储属性（但是不是那种每个对象一个数组的方式）。我们在了解它的实现前先来看一下它的用法。

要实现包含“name”、“age”和“position”属性的 `Employee` 类的话，你只需简单的继承 `ObjectTemplate`，并为名为 `attributes` 的静态方法（由 `ObjectTemplate` 输出）提供一个属性名列表，如下所示：

```
package Employee;
use ObjectTemplate;           # 加载 ObjectTemplate
@ISA=qw(ObjectTemplate);     # 继承它
attributes qw(name, age, position); # 声明你的属性
```

---

注 1: 我原来向 `comp.lang.perl.misc` 新闻组中投递了这种方案的试用版，名叫 `ClassTemplate` 模块。这里出现的版本有了显著的改进。

就这么简单。现在，模块的使用者就可以使用一种名为new的动态生成方法，来创建Employee对象，然后用存取方法（也是自动创建）来获取和修改对象属性：

```
use Employee;
$obj = Employee->new(
    "name"=>"Norma Jean",
    "age" =>25
);      #new()由ObjectTemplate创建
$obj->position("Actress");
print $obj->name, "："; $obj->age, "\n";
```

注意，Perl允许你在不存在用法歧义性的情况下，省略任何调用方法的括号。与前面的例子一样，任何跟在箭头后面的单词都被当作是一个方法。

ObjectTemplate 为一个继承类提供如下的功能：

1. 一个名为new的空间分配函数。它将为bless到继承类中的对象分配空间。new调用initialize，而后者反过来可以在继承类中被重载，以前已经讲过。
2. 与属性拥有相同名称的存取方法。这些方法将在继承类中创建，包括对象自己的方法，任何人都只能通过这些方法来存取对象属性。这是因为ObjectTemplate是唯一知道属性是如何存储的模块。例如，

```
package Employee;
sub promote {
    my $emp=shift;                #$emp 包含这个对象
    my $current_position=$emp->position()    ;# 获取属性
    my next_position =lookup_next_position($current_position);
    $emp->position($next_position);        # 设置属性
}
```

3. 用户包可以与上面一样，依照相同的命名习惯来创建自己的定制存取方法；在当前的情况中，ObjectTemplate并没有自动的产生一个。如果一个定制的存取方法要访问由ObjectTemplate管理的属性的话，它可以使用get\_attribute和set\_attribute方法。
4. new()需要一个初始化列表，一组属性名字值对。
5. ObjectTemplate考虑到属性继承(@ISA)的问题，包括内存布局和存取方法。想一想下面的例子：

```

package Employee;
use ObjectTemplate;
@ISA=qw(ObjectTemplate);
attributes qw(name age);

package HourlyEmployee;
@ISA=qw(Employee);
attributes qw(hourly_wage);

```

在这个例子中，类HourlyEmployee的对象将包含两个继承过来所有雇员所共有的属性：name和age，和只有小时雇员才有的hourly\_wage属性。

6. 所有的属性都是标量变量类型，因此friends这样的拥有多个值的属性将会以引用形式来存储：

```

attributes qw(friends);
$obj->friends(['Joe']);          # 传递给存取方法一个指向数组的引用

```

对于散列表来说当然也是如此。

## ObjectTemplate 的内部工作情况简介

图 8-1 描述了 ObjectTemplate 中对象属性是如何组织的。

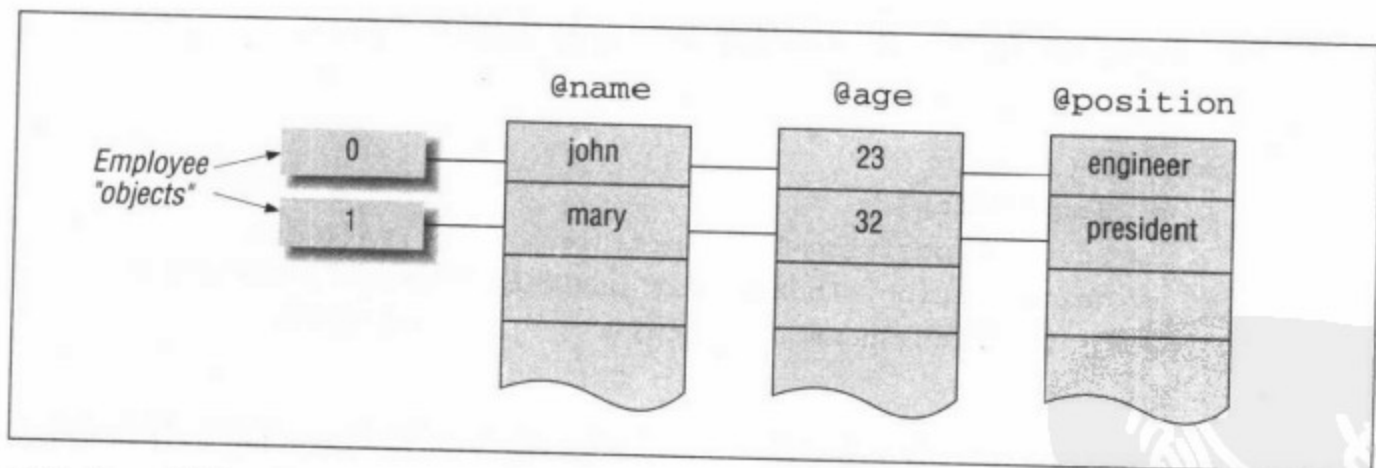


图 8-1 ObjectTemplate 的属性存储模式

数据结构非常简单。ObjectTemplate 没有为每个对象分配一个数组或散列表，而是有多少属性就分配多少数组（图中的列表示数组），而每个对象只是一条跨越数组列的横向的切片。当调用new()时，它分配一个新的逻辑行，并将初始化数组的每个元素以新的行偏移量插入到相应的属性列中。因此，这个“对象”只不过是包含行索引的经过bless的标量变量。这种模式要比散列表的空间效率更高，

因为它只创建了这么少的容器数组（只和属性的数量一样多），而且速度更快，因为存取数组要比存取散列表快一些。

在对象被删除时会有一点点麻烦。尽管相应的逻辑行被释放了，但是我们实际上无法将下面的行往上提，这是因为那样的话，其他对象的引用（它们是一些索引）和数据将会变得不一致。因此，ObjectTemplate 通过为每个包维护一个称做 `@_free` 的“自由列表”来重用被释放的（自由的）行。这是一种包含所有空闲行的链表，由一个名为 `$_free` 的标量变量指向链表的头部。表中的每个元素包含下一个空闲行的索引。当一个对象被删除时，`$_free` 将指向这个被释放的行，而空闲链表中相应索引将指向原先有 `$_Free` 所指向的前一个条目。

既然被释放的和活动的行不会重叠，因此我们可以自由的使用其中的一个属性列来保存 `@_free`。这是通过 `typeglob` 别名机制来实现的。图 8-2 描述了该结构某一时刻的状态。

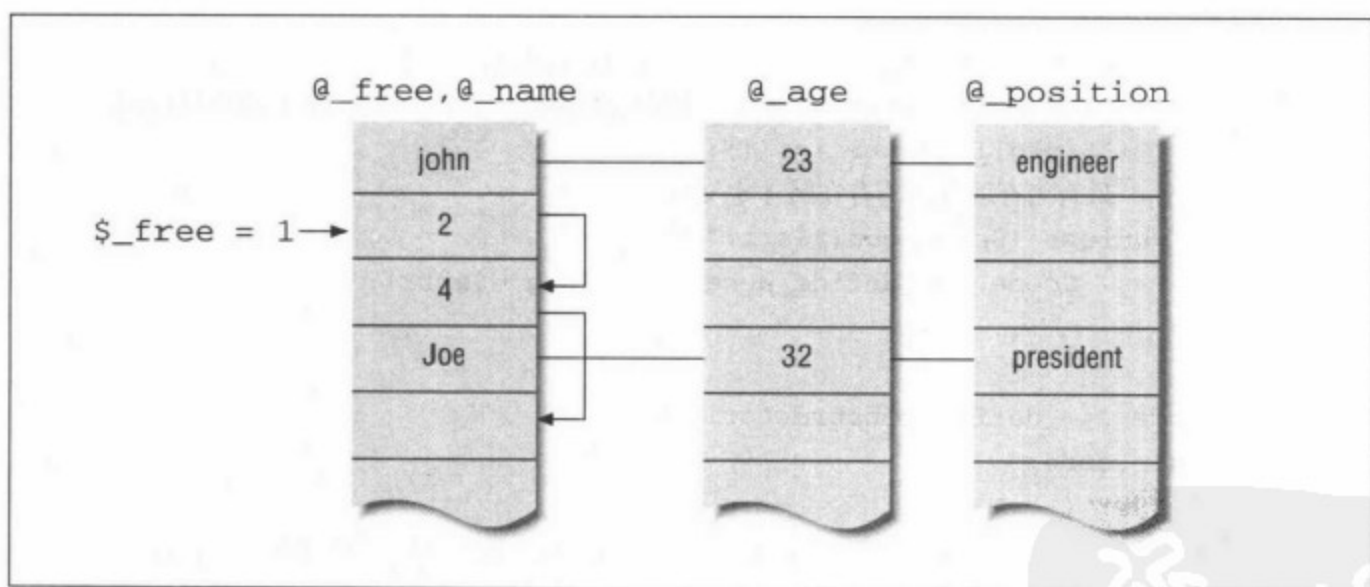


图 8-2 ObjectTemplate 用以管理由释放对象所产生的空洞的模式

你也许注意到了我使用同样的标识符名 `_free` 来表示两个变量 `$_free` 和 `@_free`。尽管我总的来说并不赞成这种方法，但是用在这里有两个理由。首先，它们两个都是为了完成相同的任务；第二个就是一个 `typeglob` 别名就可以使我们同时存取两个变量。我们马上就要看到，这对于提高性能是很重要的。

## ObjectTemplate 的实现

ObjectTemplate 中大量应用了对象、typeglob 别名、符号引用和 eval 技术。因此如果你能够理解下面的代码，那么你就可以算得上是一位 Perl 黑客了！通读这段代码的一种方法就是阅读本节所提供的描述信息，并同时使用调试器单步执行使用这个模块的一个小例子。当然，你无须理解这段代码就能够使用它。

```
Package ObjectTemplate;
Require Exporter;
@ObjectTemplate::ISA=qw(Exporter);
@ObjectTemplate::EXPORT=qw(attributes);

my $debugging=0;                                # 当赋值为 1 时可以看到凭空产生的代码

# 创建存取方法和 new()
sub attributes {
    my ($pkg) = caller;
    @{"${pkg}::_ATTRIBUTES_"} = @_;
    my $code = "";
    foreach my $attr (get_attribute_names($pkg)) {
        # 如果域名为 "color", 那么就在调用包中创建一个名为 @_color 的全局数组
        @{"${pkg}::_$attr"} = ();
        # 只有当存取方法不存在时才定义它
        unless ($pkg->can("$attr")) {
            $code .= _define_accessor ($pkg, $attr);
        }
    }
    $code .= _define_constructor($pkg);
    eval $code;
    if ($?) {
        die "ERROR defining constructor and attributes for   pkg   "
            . "\n\t${pkg}\n"
            . "-----"
            . $code;
    }
}
```

`attributes` 通过使用符号，引用来创建一个用以记录属性名的名为 `@_ATTRIBUTES_` 的全局数组。该数组将被 `get_attribute_names` 用来定义在当前包中和它的所有超类中的所有属性。对于每一个这样的属性，`attributes` 在当前包中创建一个全局数组，正如我们在图 8-1 中所看到的那样。如果还没有为那

个属性定义存取方法，它将调用 `_define_accessor` 来动态产生这个方法。最后，它调用 `_define_constructor` 直接的在调用包中创建子例程 `new`。

```
sub _define_accessor {
    my ($pkg, $attr) = @_;

    # 这段代码为给定的属性名创建存取方法
    # 这个方法在没有参数的情况下返回属性的值，如果有参数则修改属性值
    # 任何一种情况下都会返回属性最新的值
    # qq 可以使这段代码像双引号括起来的字符串

    my $code = qq{
        package $pkg;
        sub $attr {
            \@_ > 1 ? \$_{$attr} \[\$_{\$_[0]}] = \$_[1] # 存取方法 ...
                : \$_{$attr} \[\$_{\$_[0]}];             # 设置
                                                         # 获取
        }
        if (!defined \$_free) {
            # 为第一个属性列创建别名 _free
            \*_free = \*_{$attr};
            \$_free = 0;
        }
    };

    $code;
}
```

提供给 `attributes` 的每一个字段名和模块超类中的每个属性均会调用 `_define_accessor`。例如，对于 `Employee` 模块中的一个名为 `age` 的属性，它会产生如下代码：

```
package Employee;
sub age {
    @_ ? $_age[$$_[0]] = $_[1];
        : $_age[$$_[0]];
}
if (!defined $_free) {
    *_free = *_age; # 为第一个属性列创建别名 _free
    $_free = 0;
};
```





`$_[0]`中包含对象，而`$_[1]`中包含属性值。因此`$_[0]`中将包含行索引，而`$_age[$_[0]]`中将包含那个对象属性`age`的值。而且，如果别名不存在的话，`_define_accessor`将为`_free`创建别名`_age`。

```
sub _define_constructor {
    my $pkg = shift;
    my $code = qq{
        package $pkg;
        sub new {
            my \ $class = shift;
            my \ $inst_id;
            if (defined(\ $free[\ $free])) {
                \ $inst_id = \ $free;
                \ $free = \ $free[\ $free];
                undef \ $free[\ $inst_id];
            } else {
                \ $inst_id = \ $free++;
            }
            my \ $obj = bless \ \ $inst_id, \ $class;
            \ $obj->set_attributes(\ @_ ) if \ @_;
            \ $obj->initialize;
            \ $obj;
        }
    };
    $code;
}
```

`_define _constructor`为将要安装到调用包中的构造函数`new`生成所需代码。`new`将查找空闲链表，如果发现有空闲的行，它就会使用该链表中最顶层的那个行号。然后`undef` 链表头部，因为空闲链表是第一个属性列的别名，而且我们不希望属性的存取方法取到一些垃圾值。如果空闲链表中没有任何空闲的行，那么对象将会被分派给下一个逻辑索引。

```
sub get_attribute_names {
    my $pkg = shift;
    $pkg = ref($pkg) if ref($pkg);
    my @result = @{"$pkg>::_ATTRIBUTES_"};
    if (defined (@{"$pkg>::ISA"})) {
        foreach my $base_pkg (@{"$pkg>::ISA"}) {
            push (@result, get_attribute_names($base_pkg));
        }
    }
}
```



```

    @result;
}

```

get\_attributes\_names 递归的操作包的 @ISA 数组来获取所有的属性名。这可以被任何需要元数据（诸如对象持续存储模块）的人来使用。

```

# $obj->set_attributes (name => 'John', age => 23);
# 或者, $obj->set_attributes (['age', 'name'], [23, "sriram"])
sub set_attributes {
    my $obj = shift;
    my $attr_name;
    if (ref($_[0])) {
        my ($attr_name_list, $attr_value_list) = @_;
        my $i = 0;
        foreach $attr_name (@$attr_name_list) {
            $obj->$attr_name($attr_value_list->[$i++]);
        }
    } else {
        my ($attr_name, $attr_value);
        while (@_) {
            $attr_name = shift;
            $attr_value = shift;
            $obj->$attr_name($attr_value);
        }
    }
}

```

set\_attributes 将会得到一个属性的名字-值列表，然后对每一个属性都简单的调用它的存取方法。它也可以以两个参数的形式被调用，一个名字数组和一个值数组。

```

# @attrs = $obj->get_attributes (qw(name age));
sub get_attributes {
    my $obj = shift;
    my (@retval);
    map $obj->$_(), @_;
}

```

get\_attributes 使用 map 来迭代访问所有的属性名，每次迭代中将 \$\_ 设置为相应的名字。map 的第一部分只是通过符号引用来调用相应的存取方法。由于一些古怪的优先级问题，你不能够省略 \${\_} 中的大括号。

```

sub set_attribute {
    my ($obj, $attr_name, $attr_value) = @_;
    my ($pkg) = ref($obj);
    ${"${pkg}::_$attr_name"}[$$obj] = $attr_value;
}
sub get_attribute {
    my ($obj, $attr_name, $attr_value) = @_;
    my ($pkg) = ref($obj);
    return ${"${pkg}::_$attr_name"}[$$obj];
}

```

get\_attribute和set\_attribute这一对方法用来更新单一的属性值。与前面提到的那对方法不同，它们并不调用存取方法；它对属性直接更新。我们前面看到attributes不会为那些已经存在的创建存取方法。但是如果定制的存取方法还是想要使用由ObjectTemplate提供的存储模式的话，它们可以使用get/set\_attribute对。表达式\${"\${pkg}::\_\$attr\_name"}表示相应的列属性，而\$\$obj表示逻辑行。（回想一下对象只是一个指向数组索引的引用。）很明显这些方法不会有生成的存取方法速度快，因为它们使用了符号引用（它包括对字符串中变量的替换和额外的散列表查找）。

```

sub DESTROY {
    # 将id号释放回空闲链表
    my $obj = $_[0];
    my $pkg = ref($obj);
    local *_free = *{"${pkg}::_free"};
    my $inst_id = $$obj;
    # 释放行中所有的属性
    local(@attributes) = get_attribute_names($pkg);
    foreach my $attr (@attributes) {
        undef ${"${pkg}::_$attr"}[$inst_id];
    }
    $_free[$inst_id] = $_free;
    $_free = $inst_id;
}

```

DESTROY释放与对象相关的所有属性值。这项操作是必要的，因为对象只是指向一个数组索引的引用，而当它被释放时，不会改变任何属性的引用计数。一个定义自己的DESTROY方法的模块必须确保总是调用ObjectTemplate::DESTROY。

```
sub initialize ( );      # 如果子类不定义一个的话，这只是个哑方法
```

如果模块想要做一些特定的初始化工作来补充生成的`new()`所自动完成的工作的话，则需要重载这个方法。

### 建议对 ObjectTemplate 的改动

有两个方面(至少)可以进行相当大的改进。一个是`get_attributes`和`set_attributes`较慢，这是因为它们总是调用存取方法，即便是知道哪一个方法是人工提供的。由于`set_attributes`由自动生成的`new`来调用，它极大的降低了对对象的构造速度。(在没有参数的情况下，调用`new`同分配一个匿名散列表一样快。但是在调用`set_attributes`之后，它就大约慢了三倍。)

第二，定制的存取方法速度很受影响，这是因为它们不得不调用另一对缓慢的子例程`get_attribute`和`set_attribute`。或许另一种更好的方式是动态产生带有前缀“\_”的存取方法，这样开发人员就能够编写正常的存取方法了(不带前缀)并且还可以调用这些私有方法。

你也许想查看一下 CPAN 上的 `MethodMaker` 模块和标准发行版所带的 `Class::Template` 模块。这些模块同样也会自动创建存取方法，但是要假定对象的表示方式为散列表。如果你喜欢这些模块所提供的接口，你可以试着把它们的接口同 `ObjectTemplate` 的属性存储模式整合起来。

### 使用 typeglob 进行属性存储

我们以前提到过，这种方法决不是一种可读性很好的范例，而且我们之所以在这里讲到它，仅仅是因为它应用在 CPAN 上的一些可自由获得的库中，如 `IO` 和 `Net` 的发行版了。如果你不想理解这些模块是如何工作的，你可以简单的跳过这一节而不会有任何连贯性的损失。

我们在第三章“`Typeglob` 和符号表”中学到一个 `typeglob` 包含了指向不同类型值的指针。如果我们以某种方式把一个 `typeglob` 变成对象的引用，我们就能够把这些值指针当作属性而进行快速存取。考虑下面的 `foo typeglob`。

```

$(*foo) = "Oh, my!!" ; # 使用标量变量部分来存储字符串
@(*foo) = (10, 20);    # 使用数组部分来存储数组
open (foo, "foo.txt"); # 将它用做文件句柄

```

我们能够在标识符 `foo` 上挂不同类型的值（至多每种类型一个）。如果我们想要很多这样类型的对象，则可以使用 Perl 库中的 `Symbol` 模块来创建指向动态建立的 `typeglob` 的引用：

```

use Symbol;
$obj = Symbol::gensym(); # 指向 typeglob 的引用

```

`$obj` 中包含了一个指向 `typeglob` 的引用。一个 `typeglob` 的不同部分可以被分别存取（将 `foo` 替换成 `$obj` 即可）：

```

$(*$obj) = "Oh, my!!" ; # 使用标量变量部分来存储字符串
@(*$obj) = (10, 20);    # 使用数组部分来存储数组
open ($obj, "foo");     # 将它用做文件句柄

```

很明显，这对于一般的对象来说是一种糟糕透顶的方法；比如，如果需要另一个标量变量值的属性，你别无选择，除非将它放到这个 `typeglob` 的散列表部分。`IO` 模块组之所以使用这种技巧的原因就是任何模块的实例都能够当作文件句柄并且可以直接传递给诸如 `read` 和 `write` 的内建函数中（无须进行间接访问）。例如：

```

$sock = new IO::Socket( ... 各种参数 ... );
print $sock "Hello, are you there";
$message = <$sock>;

```

我们将在使用套接字进行网络编程的章节中大量使用 `IO::Socket` 模块（注2）。

让我们通过创建一个名为 `File` 的模块来更详细的查看一下这种技巧。这个模块允许你打开一个文件并读取下一行；而且还允许你将一行信息放回，这样下一次读取文件的操作将会返回这一行：

```

package main;
$obj = File->open("File.pm");
print $obj->next_line();

```

---

注2： 你不必了解下面的技术或 `IO::Socket` 模块是如何构建的就能使用它。



## 代理

代理 (delegation) 是指这样一种技术，一个对象可以借此将方法调用转发给一个指定的代理对象。在下面的例子中，类 `Employee` 简单的将所有与税务有关的功能全部移交给对象 `$accounting_dept`:

```
package Employee;
sub compute_after_tax_income {
    $me = $_[0];
    return $accounting_dept->compute_after_tax_income($me);
}
```

有时你想将所有一个类无法处理的方法调用转发给一个代理来处理。这在 Perl 中是小事一桩，因为如果一个过程在包或它的基类中找不到的话，`AUTOLOAD` 函数将会被调用:

```
package Employee;
sub AUTOLOAD {
    my $obj = $_[0];
    # $AUTOLOAD 中包含有不存在方法的名字

    # 决不要传递 DESTROY 方法
    return if $AUTOLOAD =~ /::DESTROY$/;

    # 去掉它前面的包名 (如 Employee::)
    $AUTOLOAD =~ s/^.*:://;
    $obj->{delegate}->$AUTOLOAD(@_);      # 注意 $obj 仍是 @_ 的一部分，
                                           # 因此代理函数知道原先的目标
}
```

注意，如果 `DESTROY` 没有定义，`AUTOLOAD` 将会被执行，而且重要的一点是你不能转发那条消息，否则代理将会认为 Perl 要销毁它并提前释放它的资源。

这种技术通常应用在客户/服务器库的核心部分。在一个典型的客户/服务器系统中，服务器端拥有“真正”的对象。但是系统被编写成这样一种模式，那就是一个客户端可以使用熟悉的面向对象语法调用那个对象的方法。例如，如果一个客户端程序想要调用远端一个银行帐户对象的方法，它可以这么来写:

```
$account->deposit(100);      # 存 100 块
```

表面上它似乎象一个普通的方法调用。功能库向你隐藏了 `deposit()` 实际上是在另一台机器上完成的。这是怎么实现的呢？其实 `$account` 对象的引用指向一个位于客户端的轻量级的代理对象。它主要的目的就是为调用转发给远程的机器（比如通过套接字来发送信息）并等待响应的到来。换句话说就是 `account` 对象并不是真正的帐户。它只是一个消息转发器。在底层消息系统的帮助下，它的功能移交给远程对象来完成。

## 关于继承

我的确从来没有对使用继承（inheritance）感到十分的满意，而且我并不认同这种功能对于软件重用是必需的理论。存在三种相关但不同的继承。在这一节我将会列出这些方面中我喜欢的和不喜欢的。这三种继承类型如下：

- 属性继承（attribute inheritance）
- 实现继承（implementation inheritance）
- 接口继承（interface inheritance）

## 属性继承

这种由语言提供的允许子类从基类中继承属性或结构的机制，被称作属性继承。虽然 C++ 和 Java 提供这种机制，Perl 则没有。Perl 程序员担负了设计一种让子类和基类认同的通用继承表达形式的责任。由于这个原因，散列表是经常的选择，但是正如前面已经提到的，这未必是一种经济的方法。

我对使用属性继承的疑问是，它引入了继承类（inherited class）与导出类（derived class）之间的大量纠葛。一项在基类中的改动对导出类有着极大的影响。这显然是违反封装的原则的。C++ 默认将所有的属性设置为私有的，但是还提供了一个叫做“protected”的关键词，通过使用它，这些属性可以为导出类自由使用，但对于外界依然是隐藏的。Bjarne Stroustrup, C++ 的创始人在他的优秀著作《The Design and Evolution of C++》中对此表示懊悔：

我对protected担心的一点，恰恰就是人们会，滥用全局数据那样，很容易使用一个公共的基础。回想起来，我觉得protected就是一个典型案例，说明了“好的理由”与时尚战胜了自我判断与接受新功能的经验法则。

一种更好的选择就是提供存取方法并依赖于接口继承。我们马上就要更多的谈到这一点。

## 实现继承

Perl 只支持这种类型的继承。实现继承与属性继承一样会强迫基类和继承类拥有对对象属性布局的共同理解；在使用实现继承时我们几乎总是需要属性继承。

设计子类并不容易，正像 Erich Gamma 等在《Design Pattern》一书中所谈到的那样：

设计子类同样要求对父类的深刻理解。例如，重载一个操作或许会要求重载另外一个操作。一个重载操作或许会需要调用一个继承的操作。而且子类的继承会导致类爆炸，因为你可能为了一个简单的扩展而引入许多新的子类。

他们建议使用组合来代替，我们马上就要谈到这一点。

## 接口继承

属性继承和实现继承都是为了对象实现者的方便。接口则是为了包的使用者。Perl 只支持实现继承。

一组公共可用的方法定义了对象的接口。一个导出类可以通过添加新的方法来增加新功能。但是它是否真的重载了基类的实现，严格来说是实现的问题；从用户的角度来说，它仍然提供同样的方法。

有关接口的重要一点就是它表示了用户和对象间的一种协议。如果两个对象拥有相同的接口，那么它们可以互换使用。在这方面，替换表现了一种语言或一组部件所能提供的最重要的功能。



## 使用组合来代替

当我为 Xt/Motif (X Window 平台的 GUI 框架) 编写一些组件的时候, 自己就曾确信实现继承是必要的。这个框架使用 C 语言, 并在很大程度上提供了单重继承的特性, 但其结果却并非容易使用。在 C++ 出现的时候, 我很快就对这种支持继承的语言热情高涨, 并试图以 C++ 来实现这个组件集。然后在 John Ousterhout 编写的 Tk 出现时, 我那种创建组件的简便方式感到惊叹, 尽管它是用 C 语言编写的, 但还是提供了 Motif 所提供的所有功能 (甚至还要多得多)。Tk 使用了组合的体系结构, 却不是继承。因此我适当纠正了自己的观点。

组合 (composition) 的思想就是一个对象可以由其他对象组合而成。也就是说, 它与其他类之间体现的是“具备”或“使用”的关系, 而不是“就是”的关系。许多出版文献的例子中都极力宣扬实现继承, 但结果它们只不过是组合的优秀候选对象。考虑一下这个常常被当作例子来举的名为 Vice-President (副总裁, 简称 V.P.) 的类, 它继承自名为 Manager 的类, 而 Manager 又继承自类 Employee。不错, 一个 V.P. “就是”一个 Manager, 而它依次又是一个 Employee, 因此这就是属性与实现继承里的情况。但是如果雇员被晋级会发生什么呢? 该对象就会被迫改变它的类——这是显然是一种糟糕的设计。一种更好的解决方式就是实现一个在公司中扮演一个或多个角色的雇员 (可以为一个 Manager, vice-president 或 lead technical engineer), 而且当该雇员晋级时, 更新的只是它所担当的角色。换句话说就是, Employee 对象“使用”类 Role, 而就它而言则囊括了有关角色的一切, 如工作描述、薪金范围、以及先决条件等。

组合也被称做组件驱动 (component-driven) 的编程。开发可重用软件的关键, 就是开发具有定义良好和文档齐全的接口的完全封装起来的组件。就我的经验而言, 考虑继承性的设计, 很少会产生像所宣传的那样的好处。

Perl 提供了用于创建即插即用组件的最为关键的特性: 多态性和运行时联编。你可以编写 `$obj->draw()`, 而 Perl 会根据对象所属的类来恰当的调用方法 `draw()`。由于 Perl 为隐含式类型语言, 因此这条语句对于不管是图形、枪炮或六合彩都能工作。我觉得这项功能的意义要远高于其对实现继承的支持。

## 相关资源

1. Design Patterns: Elements of Reusable Object-Oriented Software. Erich Gamma, Richard Helm, Ralph Johnson 和 John Vlissides. Addison-Wesley, 1994。
2. The Design and Evolution of C++. Bjarne Stroustrup. Addison-Wesley, 1994。



#### 本章简介:

- 标量变量的绑定
- 数组的绑定
- 散列表的绑定
- 文件句柄的绑定
- 例子: 对变量的监控
- 与其他语言的比较

## 第九章 绑定

兄弟, 给我一条激情似火的领带, 一条带有  
无限力量的领带, 一条会起誓与撕扯的领带,  
每当它配上我那套老式的蓝色哔叽西服时。

—— Stoddard King

打好的那条领带

通常情况下, 当你读或更新一个标量变量、数组、散列表或文件句柄时, Perl 就会在内部数据结构上执行相应的操作。另外, 你还可以使用关键词 `tie` 将值 (或变量) 绑定到用户定义的对象上, 这样当你从变量中读或写时, Perl 就会简单的调用它所绑定的对象的特定方法。换句话说, Perl 尽管为“普通”变量提供了相应的实现, 但是它还可以允许由用户定义的模块来为绑定的变量完成这些工作。一旦变量被绑定, 即便是通过 Perl 库中的 C API 中来存取也会委托给相应的绑定对象来完成。

这种方法似乎有些像语法游戏, 但是正如你将会从本章的例子中看到, 这种语法提供了强大的功能: 一个普通的变量可以在任何被操作的时候调用一个用户定义的函数而无须改变用户的代码, 甚至未必能意识到这种幕后的操作。这种技术最常见的用途就是将一个散列表变量绑定到一个可以操纵 DBM 文件的模块上, 这些文件为典型的基于磁盘的散列表 (它们也可以是 B 树)。该技术可以使你将一个散列表值进行持续性存储, 而且能够保存比内存多得多的信息, 同时还会给你一种操作普通关联数组的印象。

下面我们将学习 `tie` 在各种数据类型中是如何工作的, 并且会看到应用此项功能的几个有用的例子。

## 标量变量的绑定

从最基本的层次上来讲，对标量变量只有四种操作。你可以创建它，获取或设置它以及销毁它（通过结束作用域或使用 `undef`）。`tie` 允许你为每一种操作提供相应的用于回调的子例程。

`tie` 的语法如下：

```
tie 变量, 类名, 列表;
```

第一个参数应当为上面描述的四种类别中的一种。第二个参数为一个用户定义类名。你应当在调用 `tie` 之前已经调用了 `use` 类名或 `require` 类名。

在这条语句执行时，Perl 会检查变量的类型（第一个参数）。然后它将根据绑定变量的类型、标量变量、数组、散列表或者文件句柄来相应的调用方法 `classname->TIESCALAR(list)`、`TIEARRAY`、`TIEHASH` 或 `TIEHANDLE`。如果这种方法不存在的话将会导致运行时错误。`TIESCALAR` 返回一个对象，而该对象又在内部同给定的变量联系起来（或称做绑定）。现在当你读或写这个变量时，Perl 在内部将会分别调用 `object->FETCH()` 和 `object->STORE()`。最后当绑定的变量超出作用域时，Perl 将会调用 `object->DESTROY()`。很简单，是不是？

名字 `FETCH`、`STORE` 和 `TIESCALAR` 同 `AUTOLOAD` 和 `DESTROY` 相似，它们只有在适当的环境中才会拥有对 Perl 特殊的意义。也就是说，一个模块可以有一个名为 `FETCH` 的方法，它可以像任何其他用户定义的子例程一样来正常的使用。但是如果你使用了 `tie`，这个方法则会被认定具有了一种特殊的含义。

Perl 并不关心对象使用的确切数据结构（不管你使用的是散列表还是 `ObjectTemplate`）。表 9-1 描述了一个绑定到自动温度控制系统上的变量，该控制系统以一个名为 `AC.pm`（注 1）的 Perl 模块来表示。一个试图对 `$temperature` 的读操作将会被翻译成一个对温度传感器的调用，而试图设置变量值的操作将会被翻译成一条命令，指示加热系统完成所需任务。

---

注 1：是空调，可不是交流电！

表 9-1 一个标量变量绑定操作中的控制流

当你这么写时:	Perl 将它翻译成方法调用:	方法的内容:
<code>tie \$temperature,     'AC';</code>	<code>\$obj = AC-&gt;TIESCALAR() Perl 现在绑定 \$temperature 和 \$obj</code>	<pre>package AC; sub TIESCALAR {     ...     \$ac = bless {...},     \$pkg;     return \$ac; }</pre>
<code>\$x = \$temperature;</code>	<code>\$x = \$obj-&gt;FETCH();</code>	<pre>sub FETCH {     \$ac-&gt;get_temp(); }</pre>
<code>\$temperature = 20;</code>	<code>\$obj-&gt;STORE(20);</code>	<pre>sub STORE {     (\$obj, \$t) = @_;     \$ac-&gt;set_temp(\$t); }</pre>
<code>untie \$temperature; 或 undef \$temperature; 或当 \$temperature 超出范围时</code>	<code>\$obj-&gt;DESTROY()</code>	<pre>sub DESTROY { }</pre>

正如你所看到的，AC 模块是带有一个构造方法和三个对象方法（而它们碰巧拥有特殊的名字）的普通的类。Perl 在幕后就是与这个模块进行交互，为用户提供了一种更为简单交互模式。你可以从 `tie` 的返回值中获取绑定对象，或是在任何时候调用 `tied` 函数来访问到它。因此语句：

```
$temperature=20;
```

等价于下面的语句：

```
(tied $temperature)->STORE(20);
```

`untie` 函数将恢复变量原来的值，同时还调用对象的 `DESTROY` 方法。

Perl除了要求对象模块提供我们前面见到的方法外,并不对其有任何限制。它可以存储任意想要的数,可以拥有其他的方法而且甚至还可以很好的工作在非tie的环境中。

## 例子: 秒表

让我们来看一个应用标量变量绑定的秒表的例子。当你向它存储数据时,它会记下当前的时间(也就是说它会忽略当前值)。当你从其中获取数值时,它会返回自从上一次存储操作以来流逝的秒数。下面是它的用法:

```
use Stopwatch;
tie $s, 'Stopwatch';

# $s 以标量变量的形式透明的绑定到一个秒表对象
$s = 0;                # 对它的写操作将迫使它复位
sleep(10);              # 睡 10 秒钟
print "$s\n";          # 应当打印出 10
```

由于sleep的分辨率,这个例子或许有时会打印出结果9。

例9-1: 使用tie来实现秒表

```
package Stopwatch;

sub TIESCALAR {
    my ($pkg) = @_;
    my $obj = time(); # $obj 中保存了上次复位时的时间
    return (bless \$obj, $pkg);
}

sub FETCH {
    my ($r_obj) = @_;
    # 返回自从上次复位时流逝的时间
    return (time() - $$r_obj);
}

sub STORE {
    my ($r_obj, $val) = @_;
    # 忽略这个值。任何对它的写操作将被视为复位
    return ($$r_obj = time());
}
```

```
1;
```

TIESCALAR记下当前的时间，并返回一个经过bless的指向标量变量的引用（其中包含当前时间）。正如我们前面所提到的那样，你没有义务必须提供一个经过bless的标量变量引用；Perl并不关心这个对象是标量变量还是数组或是一种复杂的数据结构。唯一的要求就是将其bless到一个支持FETCH和STORE方法的模块中。在本例中，FETCH计算出当前时间（由time提供）同上次复位时间之间的间隔时间。

顺便说一下，该模块中的时间计算工作处在秒一级的精度。如果你需要更好的精度，你可以使用能够从CPAN获得的Time::HiRes模块，它在Unix系统中提供毫秒一级的精度（提供对usleep和ualarm系统调用的存取）。在微软Windows系统中，你可以使用Win32::Timer包来获得毫秒级的计时。

### tie 能够与匿名值一起工作

tie的第一个参数必须化归为一个标量变量、数组、散列表或文件句柄值；它不一定要是一个变量。下面的代码描述了两个有效的标量变量绑定的例子：

```
$r = \%s;  
tie $$r, 'Stopwatch'; # 对 $s 的间接绑定  
  
@foo = (1, 2);  
tie $foo[1], 'Stopwatch';
```

正如你所看到的，这种机制需要的是内在的值，而并没有牵连到变量名（这与Tcl中的trace机制不同）。

## 数组的绑定

将数组绑定到一个模块工作的也非常相似，如表9-2中所示。你可以在两个层次上操作一个普通的数组。一个就是你可以获取和设置整个数组的值以及最后一个元素的索引（使用\$array）。另一个就是你可以获取或设置单一的元素，并使用splice、push、pop等来创建或销毁其中的元素。在这本书出版印刷时，tie

只处理对数组元素的读和写，并不允许对整个数组做任何改动。这种状况估计在不远的将来会得到改善。

表 9-2 数组的绑定与存取

当你这么写时:	Perl 将它翻译成:
<code>tie @array, 'Foo', 1, 2</code>	<code>\$obj = Foo-&gt;TIEARRAY (1, 2);</code>
<code>\$a = \$array[5];</code>	<code>\$obj-&gt;FETCH(5);</code>
<code>\$array[5] = "aa"</code>	<code>\$obj-&gt;STORE(5, "aa");</code>
<code>untie @array;</code>	<code>\$obj-&gt;DESTROY();</code>

绑定数组的一个有用的例子就是仿真位集。如果你将第 200 个元素设置为 1，那么模块则能够通过使用 `vec()` 来设置位数组中的第 200 个位。

下一节将描述一个包裹文本文件的绑定数组的例子。

## 绑定数组的例子：把文件当作数组

这个例子将创建一种名为 `TieFile` 的机制，它可以使一个文件看起来像数组。例如，如果你想检查文件 `foo.txt` 的第 20 行的话，你可以这么写：

```
tie @lines, 'TieFile', 'foo.txt';
print $lines[20];
```

为了简单起见，该模块并不接受对任何元素的更新。

当需要获取第  $n$  行时，例 9-2 中的 `TieFile` 模块会一直读到文件的这一行并将其返回。由于在每次申请一行信息时总是遍历整个文件有点儿多余，`TieFile` 将保留每一行的开始所处的文件偏移量，这样当你申请到它曾经访问过的一行信息时，它会知道该行的确切偏移量并在读之前使用 `seek` 定位到这个位置。由 `TIEARRAY` 创建的对象有两个字段：一个用以存储一组偏移量信息而另一个用以存储打开文件的文件句柄。这两个字段均保存在匿名数组中。（另外你还可以使用散列表或 `ObjectTemplate` 模块。）



例 9-2: TieFile.pm:将文件映射到数组中

```
package TieFile;
use Symbol;
use strict;
# 在 TIEARRAY 中构建的对象是一个数组
# 而这些又是它的字段
my $F_OFFSETS = 0; # 文件偏移量列表 (对每一行来说)
my $F_FILEHANDLE = 1; # 打开文件句柄

sub TIEARRAY {
    my ($pkg, $filename) = @_;
    my $fh = gensym();
    open ($fh, $filename) || die "Could not open file: $!\n";
    bless [ [0], # 第 0 行的偏移量为 0
            $fh
          ], $pkg;
}

sub FETCH {
    my ($obj, $index) = @_;
    # 我们是否已经读过这一行了呢?
    my $rl_offsets = $obj->[$F_OFFSETS];
    my $fh = $obj->[$F_FILEHANDLE];
    if ($index > @$rl_offsets) {
        $obj->read_until ($index);
    } else {
        # 移动到相应的文件的便移量位置
        seek ($fh, $rl_offsets->[$index], 0);
    }
    return (scalar <$fh>); # 通过计算 <$fh> 返回一行信息
                          # 上下文为标量变量
}

sub STORE {
    die "Sorry. Cannot update file using package TieFile\n";
}

sub DESTROY {
    my ($obj) = @_;
    # 关闭文件句柄
    close($obj->[$F_FILEHANDLE]);
}
```

```

sub read_until {
    my ($obj, $index) = @_;
    my $rl_offsets = $obj->[$F_OFFSETS];
    my $last_index = @$rl_offsets - 1;
    my $last_offset = $rl_offsets->[$last_index];
    my $fh = $obj->[$F_FILEHANDLE];
    seek ($fh, $last_offset, 0);
    my $buf;
    while (defined($buf = <$fh>)) {
        $last_offset += length($buf);
        $last_index++;
        push (@$rl_offsets, $last_offset);
        last if $last_index > $index;
    }
}

1;

```

你也许已经注意到了, 这个模块只有在你赋给绑定的数组元素字符串和数字时才能工作。如果你赋值给它一个引用, 那么它就会简单的将其转换成字符串然后存储到文件中, 很明显, 再次从文件中读回的数据将没有任何意义。换句话说, 这个模块应当在存储到文件之前将由引用指向的数据结构进行理想的“序列化”, 并在需要时重新创建它。我们将在第十章“持续性”中更多的谈到这个问题。

## 散列表的绑定

与数组不同, 对绑定的散列表的存取得到了全面的支持。散列表的绑定机制可以使你捕获对整个散列表的操作, 对单一元素的存取, 以及查询操作 (exist、defined、each、keys 和 values) 等。表 9-3 描述了这些动作是如何映射到对绑定对象方法的调用上的。

表 9-3 绑定与散列表存取

当你这么写时:	Perl 将它翻译成:
tie %h, 'Foo', 'a'=> 1	\$obj = Foo->TIEHASH('a',1);
\$h{a}	\$obj->FETCH ('a')
\$h{a} = 1	\$obj->STORE ('a', 1)

表 9-3 绑定与散列表存取 (续)

当你这么写时:	Perl 将它翻译成:
<code>delete \$h{a}</code>	<code>\$obj-&gt;DELETE('a')</code>
<code>exists \$h{a}</code>	<code>\$obj-&gt;EXISTS('a')</code>
<code>keys (%h), values (%h)</code>	<code>\$lk = \$obj-&gt;FIRSTKEY ();</code>
<code>each (%h)</code>	<code>do {</code> <code>  \$val = \$obj-&gt;FETCH(\$lk);</code> <code>} while (\$lk = \$obj-&gt;NEXTKEY(\$lk));</code>
<code>%h = ()</code>	<code>\$obj-&gt;CLEAR()</code>
<code>%h = (a=&gt; 1)</code>	<code>\$obj-&gt;CLEAR()</code> <code>\$obj-&gt;STORE('a', 1)</code>
<code>untie %h</code>	<code>\$obj-&gt;DESTROY()</code>

FIRSTKEY 和 NEXTKEY 要求返回序列中的下一个键值。如果调用代码调用的是 keys 那么这样就能够满足要求; 但是如果调用的是 values 或 each, 那么它就会为每一个键值调用 FETCH。

Tie 最常见的用法 (也是看起来自然的用法) 就是用作 DBM 文件的前端, DBM 文件我们以前提到过, 就是基于磁盘的散列表。Perl 提供对各种 DBM 变种的支持。下面的例子中就使用了随标准 Perl 发行版一起发行的 SDBM 模块:

```
use Fcntl;
use SDBM_File;
tie (%h, 'SDBM_File', 'foo.dbm', O_RDWR|O_CREAT, 0640)
    || die $!;                # 打开 DBM 文件
$h{a} = 10;                   # 透明的向文件中写
while (($k, $v) = each %h) {  # 对文件中的所有键值进行迭代
    print "$k,$v\n"}
untie %h;                      # 刷新并关闭 DBM 文件
```

Perl 的老用户或许已经意识到了这与 dbm\_open 函数的相似之处。tie 碰巧只是一种更为通用的机制。

绑定的散列表同样会有上一节所提到的问题: 除非你显式的将引用所指向的结构序列化或流的形式 (你也可以以后从中重新创建所需的数据结构), 否则你不能存

储引用。我们将要在第十章进一步探讨的MLDBM模块会试图将多层的散列表绑定到一个 DBM 文件上。

在标准Perl发行版中还有两个模块在内部使用tie。Config模块将创建环境中(这是对configure来说的)的所有信息作为调用者名字空间中的一个散列表,例如:

```
use Config;
while (($k, $v) = each %Config) {
    print "$k => $v \n";
}
```

Env是另一个通过tie来使环境变量值看起来像普通变量的标准库。我们已经在第六章“模块”中的“符号的导入”一节看到过一个非tie的变体Env。

## 文件句柄的绑定

如表 9-4 所示,当你读或写一个文件句柄时绑定的文件句柄就会调用一个用户定义对象。注意tie语句要以一个typeglob为参数,而不是一个光杆儿单词。

表 9-4 tie 和文件句柄

当你这么写时:	Perl 会将它翻译成:
tie *FH, 'Foo', 'a', 'b'	\$obj = Foo->TIEHANDLE('a', 'b');
<FH>	\$obj->READLINE();
read (FH, \$buf, \$len, \$offset)	\$obj->READ(\$buf, \$len, \$offset)
sysread (FH, \$buf, \$len, \$offset)	
getc (FH)	\$obj->GETC();
print FH "I do"; # FH后没有逗号	\$obj->PRINT("I do");
untie *FH;	\$obj->DESTROY();

这种方法可以用于仿真带有测试模块的文件或进程,或是通过安静的记录其间的会话,来监视对文件句柄的存取。Tk通过支持tie可以让你重定向I/O到它的文本组件,有关Tk的问题我们将在第十四章“使用Tk来建立用户界面”中详细学习。

## 例子：对变量的监控

`tie`可以使你非常方便的监控变量。在这一节中，我们将开发一个名为`Monitor.pm`的模块，当你选定的变量被存取时，它会向 `STDERR` 打印出一条消息（注2）。

```
use Monitor;
monitor(\$x, 'X');
monitor(%y, 'y');
```

每当 `$x` 或 `%y` 被改变时，该模块就会在 `STDERR` 上打印出如下所示的信息：

```
Wrote   : $x ... 10
Read    : $x ... 10
Died    : $x
Wrote   : $y{a} ... 1
Cleared : %y
```

这个模块在调试时很有用，当不清楚何时某个变量发生改变，尤其是通过引用间接的进行改变时非常有用。该模块还可以被改进以支持诸如 `print 'ahhh' when $array[5]>10` 这样的监控表达式。有了 Perl 对 `eval` 的支持，这个任务相当的简单。

`monitor` 以一个变量的引用和打印输出信息时使用的名字为参数。第一个参数被用来进行变量的绑定。不幸的是 `tie` 有一个属性，那就是它会隐藏变量原先的值（这个值在 `untie` 时恢复）。很明显，我们不希望海森堡测不准原理会应用到这儿——我们对变量的监控不应当影响用户对它的观察。因此，我们将原先的值作为属性保存起来，并让 `FETCH` 和 `STORE` 来使用这个拷贝。最后，当我们不在使用这个变量时我们将会使用 `unmonitor`，它在内部调用 `untie`。

如例 9-3 所示，`Monitor` 将职责授权给一个嵌套的针对每一种类型值（标量变量，数组、散列表）的模块来完成。这些模块中的 `tie` 构造函数返回一个经过 `bless` 的匿名数组（绑定的对象），它保存了由用户提供的名字（`monitor` 的第二个参数）以及变量的当前值。

---

注2： 这是 CPAN 上一个由 Stephen Lidie 编写的称做 `Tie::Watch` 模块的轻量级版本，`Tie::Watch` 用来在特定的变量被存取时调用用户定义的回调。

例 9-3: Monitor.pm

```

#-----
package Monitor;
require Exporter;
@ISA = ("Exporter");
@EXPORT = qw(monitor unmonitor);
use strict;

sub monitor {
    my ($r_var, $name) = @_;
    my ($type) = ref($r_var);
    if ($type =~ /SCALAR/) {
        return tie $$r_var, 'Monitor::Scalar', $r_var, $name;
    } elsif ($type =~ /ARRAY/) {
        return tie @$r_var, 'Monitor::Array', $r_var, $name;
    } elsif ($type =~ /HASH/) {
        return tie %$r_var, 'Monitor::Hash', $r_var, $name;
    } else {
        print STDERR "require ref. to scalar, array or hash"
    }
    unless $type;
}

sub unmonitor {
    my ($r_var) = @_;
    my ($type) = ref($r_var);
    my $obj;
    if ($type =~ /SCALAR/) {
        Monitor::Scalar->unmonitor($r_var);
    } elsif ($type =~ /ARRAY/) {
        Monitor::Array->unmonitor($r_var);
    } elsif ($type =~ /HASH/) {
        Monitor::Hash->unmonitor($r_var);
    } else {
        print STDERR "require ref. to scalar, array or hash"
    }
    unless $type;
}

#-----
package Monitor::Scalar;

sub TIESCALAR {
    my ($pkg, $rval, $name) = @_;
    my $obj = [$name, $$rval];

```

```
    bless $obj, $pkg;
    return $obj;
}

sub FETCH {
    my ($obj) = @_;
    my $val = $obj->[1];
    print STDERR 'Read    $', $obj->[0], "... $val \n";
    return $val;
}

sub STORE {
    my ($obj, $val) = @_;
    print STDERR 'Wrote   $', $obj->[0], "... $val \n";
    $obj->[1] = $val;
    return $val;
}

sub unmonitor {
    my ($pkg, $r_var) = @_;
    my $val;
    {
        my $obj = tied $$r_var;
        $val = $obj->[1];
        $obj->[0] = "_UNMONITORED_";
    }
    untie $$r_var;
    $$r_var = $val;
}

sub DESTROY {
    my ($obj) = @_;
    if ($obj->[0] ne '_UNMONITORED_') {
        print STDERR 'Died   $', $obj->[0];
    }
}

#-----
package Monitor::Array;

sub TIEARRAY {
    my ($pkg, $rarray, $name) = @_;
    my $obj = [$name, @$rarray];
```

```

    bless $obj, $pkg;
    return $obj;
}

sub FETCH {
    my ($obj, $index) = @_;
    my $val = $obj->[1]->[$index];
    print STDERR 'Read    $', $obj->[0], "[$index] ... $val\n";
    return $val;
}

sub STORE {
    my ($obj, $index, $val) = @_;
    print STDERR 'Wrote   $', $obj->[0], "[$index] ... $val\n";
    $obj->[1]->[$index] = $val;
    return $val;
}

sub DESTROY {
    my ($obj) = @_;
    if ($obj->[0] ne '_UNMONITORED_') {
        print STDERR 'Died   %', $obj->[0];
    }
}

sub unmonitor {
    my ($pkg, $r_var) = @_;
    my $r_array;
    {
        my $obj = tied @$r_var;
        $r_array = $obj->[1];
        $obj->[0] = "_UNMONITORED_";
    }
    untie @$r_var;
    @$r_var = @$r_array;
}

#-----
package Monitor::Hash;
sub TIEHASH {
    my ($pkg, $rhash, $name) = @_;
    my $obj = [$name, (%$rhash)];
    return (bless $obj, $pkg);
}

```



```
sub CLEAR {
    my ($obj) = @_;
    print STDERR 'Cleared %', $obj->[0], "\n";
}

sub FETCH {
    my ($obj, $index) = @_;
    my $val = $obj->[1]->{$index};
    print STDERR 'Read    $', $obj->[0], "{$index} ... $val\n";
    return $val;
}

sub STORE {
    my ($obj, $index, $val) = @_;
    print STDERR 'Wrote   $' $obj->[0], "{$index} ... $val\n";
    $obj->[1]->{$index} = $val;
    return $val;
}

sub DESTROY {
    my ($obj) = @_;
    if ($obj->[0] ne '_UNMONITORED_') {
        print STDERR 'Died    %', $obj->[0];
    }
}

sub unmonitor {
    my ($pkg, $r_var) = @_;
    my $r_hash;
    {
        my $obj = tied %$r_var;
        $r_hash = $obj->[1];
        $obj->[0] = "_UNMONITORED_";
    }
    untie %$r_var;
    %$r_var = %$r_hash;
}

1;
```

unmonitor 有些棘手。我们想要进行 untie，可是 Perl 却会恢复在调用 tie 之前保存的变量值。显然这不是所期望的。我们希望这项操作能够在不受变量使用者任何影响的情况下进行工作。既然我们拥有以绑定对象属性方式存储的变量的

当前值，我们可以试图在`untie`操作之后恢复这个变量。不幸的是下面的这段代码并不能很好的工作：

```
# 对于一个绑定的标量变量
my $obj = tied $$r_var;      # 获取与变量绑定的对象
$latest_value = $obj->[1];    # 取得最新的值
untie $$r_var;               # untie
$$r_var = $latest_value;     # 恢复变量的最新值
```

如果`-w`标志置位时，Perl会打印信息：“Can't untie: 1 inner references still exist… (无法untie: 1内部引用仍然存在……)”。问题的原因就在于局部变量`$obj`增加了绑定对象的引用记数，因此`untie`操作无法 DESTROY 这个绑定的对象。解决方案相当直截了当：在内层代码块中取得变量值并让`$obj`超出作用域，如下所示：

```
my $latest_value;
{
    my $obj = tied $$r_var;
    $latest_value = $obj->[1]; # 抽取最新的值
                             # 注意 $latest_value 定义在
                             # 内层代码块的外围
}
# $obj 已经不在作用域中了，因此我们可以放心的执行 untie.
untie $$r_var;
$$r_var = $latest_value;
```

## 与其他语言的比较

我们已经以两种方法应用了`tie`机制。一个是为已经存在的包提供易用的前端（如Perl对DBM文件提供的支持）；另一个就是对现有变量的监控。让我们来查看一下别的语言在这些方面都提供些什么样的支持。

### Tcl

Tcl提供了一种称做`trace`的命令来捕获对标量变量和关联数组的存取。（标量变量与列表是可以互换的，因此对于后者没有提供单独的机制。）正如我们将在第

十四章看到的，Tk 工具包很好的利用了这种跟踪机制。trace 并不保留原先的值，因此编写一个监控包要容易些。

Tcl 的 C API 可以使你创建跟踪时要比在 Perl 中容易的多。(实际上，这种易用性同样适用于其余的 Tcl API，这一点我们将在第二十章“Perl 的内部工作”中了解到。) 尽管现存的 Tcl 包可以利用这种机制来为一个包提供易用的前端，但是我还不知道有什么包利用了这一点，而 Perl 却在 DBM 文件中加以应用了。

## Python

Python 可以允许你为每个类编写名为 `_getattr_` 和 `_setattr_` 的特殊函数，而它们则可以使你捕获对成员属性（或者模仿新的属性）的存取。类似的，你可以通过提供称做 `_getitem_` 和 `_setitem_` 的特殊方法让类来模仿数组。有 40 种方法要重载来表示所有的行为。

## C++

C++ 并不允许对变量的动态跟踪。但是另一方面，它的确提供了大量的操作符和操作符重载的语法结构，使你在基础数据类型或对象应用的地方来进行对象替换。

像诸如 Purify 这样的商业工具和功能库能够在任意区域的内存上设置动态跟踪。它们还提供了 C API 来让你对这些事件编写自己的回调函数。

## Java

Java 不允许你任意的捕获存取操作。一些商用交易处理系统甚至能够检查特定的字节代码，以识别出对成员属性的存取，并在必要的地方插入跟踪代码。这样可以无须显式的对象协作就能够操作任何对象。这种方案显然不怎么舒服。

Java 同样也没有任何办法来实现另一方面的应用：那就是使一个类看起来像普通变量。

# 第十章

## 持续性

### 本章简介：

- 有关持续性的问题
- 流数据
- 面向记录的方案
- 关系数据库

在美国肯定至少有 500 000 000 只耗子。

当然这还只是我记忆中的数目。

—— Edgar Wilson Nye

如果我们永远不用担心致命的 bug 和掉电的话，那么这的确会成为一个理想的世界（注1）。但是现在我们必须应付这样的事实，那就是电脑所能顾及范围也就是电缆的可达长度，而我们的数据太宝贵了，我们不能只将它们放在电子存储器当中。系统或模块的这种使应用系统的数据比应用执行周期要持续保留更长时间的能力，称做持续性（persistence）（注2）。

考虑到数据库意味着一个数以万亿的美元产值的工业，而 DBI（数据库接口）和相关 Perl 模块从 CPAN 的下载量仅次于 CGI 模块，说持续性是所有技术中最为重要就不算过分了。本章我们首先要学习将我们的数据进行持续性所要考虑的诸多因素；然后我们来考察一下大多数可自由获得的 Perl 持续性模块，并与这些因素进行对照，以清楚的理解它们的能力与弱点、所提供的功能，以及开发人员从什么地方开始介入。下一章我们将使用一些这样的模块来创建一个对象持续性框架，以使对象透明地存储到文件和数据库中。

注1： 或者是最终用户，正如寄给《Byte》杂志的一封信中曾经抱怨的那样。

注2： 我们将使用“系统”这个字眼来意指一种 C 实现，如 DBM 函数库或数据库等，而使用“模块”来指代 Perl 模块。

## 有关持续性的问题

数据包括从简单的以逗号分隔的记录，到复杂的自引用（self-referential）结构，用户则在不同程度的期望值和共享持续性数据的能力（和需要）上有所不同；应用编程人员试图综合各种提供不同程度简单性、健壮性和效率的方案。下面的列举就比较详细的考察了这些差异：

### 序列化

普通的数组和散列表可以用制表符、逗号等分隔形式写入到文件中。那些散列表数组或数组的数组等嵌套数据结构在写入文件之前，必须展平或序列化（serialize）。如果你曾经盘绕过假日彩灯的电线，就会明白你不仅要尽力的把它缠紧，还必须要缠得在下次使用时能轻而易举将其展开。同样，数据项可以是typeglob，还可以包含有指向原始C数据结构的指针，或是指向其他数据项的引用（以形成循环或自引用的数据结构）。在这一章我们将会学到三个进行数据序列化的模块：FreezeThaw、Data::Dumper和Storable。

### 边界

普通的文件由于是字节流，所以既不提供也不设置任何数据边界；你必须判定如何在磁盘上区分和识别每个数据项。DBM和ISAM系统安设了一种面向记录的结构。关系数据库则提供记录和列的边界。如果你的数据可以分成这种栅格结构，那就幸运一些。否则，你就碰上了常被称做“阻抗失配（impedance mismatch）”的问题。诸如对象关系型和面向对象数据库等更新的技术，将试图解决这种“限制”或“失败”的问题（注3）。

### 并发

多个应用或用户可能要对持续化数据存储进行并发的存取。一些系统完全不考虑这个问题；而另一些则提供不同类型的加锁模式。

### 存取权限

大多数持续化存储方案让操作系统来实施文件级权限的管理（创建、更新、读或删除）。数据库则提供了一种更为精细的存取限制。

---

注3：被认为是关系数据库理论之父的E.F.Codd一直称这种不匹配性并不是理论本身的内在因素；这是关系型数据库管理系统实现技术所造成的结果。

### 随机存取与插入

数据库可以使你简单的进行新记录的插入,或单一属性的更新。而对于流数据而言,你除了序列化整个数据并重新写入文件外,别无选择。

### 查询

DBM 和 ISAM 文件可以使你基于主关键字来有选择的获取记录数据,而数据库则可以使你基于任意的字段来有选择的获取记录。数据量越大,你检查每条记录符合查询条件的开销就越大。

### 事务

重要的商务应用都要求持续化存储方案提供“ACID”(译注1)属性:

**原子性 (Atomicity):** 一系列操作要么作为一个整体发生,要么什么都不做。

**一致性 (Consistency):** 事务必须使系统处于一致的状态。一致性是应用系统的责任;事务监控器或数据库对特定应用领域一无所知,所以无法判断怎么才是一致的而怎么又不是。

**隔离性 (Isolation):** 不同独立事务之间的读写操作必须相互分隔;其结果应该与系统被强制工作在串行状态时所产生的结果相同。

**持久性 (Durability):** 一旦事务完成,结果必须安全地写回磁盘。

当前只有数据库提供这样的机制,而且现有的事务型文件系统非常少。2.0版的 Berkeley DB 库提供了并发存取、事务和恢复机制,但是在编写本书时相应的 Perl 包裹函数还没有更新,无法利用这些优势。

### 元数据

如果你能够存取描述数据的信息—元数据,那么你就可以进行更少的硬编码工作。数据库提供对元数据的显式支持,而其他的方案只是简单的将磁盘数据转换成内存中的 Perl 结构,然后由 Perl 来提供元数据信息。

### 机器独立性

你也许想要获取在另一台不同类型机器上创建的文件中的数据。你必须处理不同的整数和浮点数表示形式,包括字节数量和字节顺序。

---

译注1: ACID 就是 Atomicity, Consistency, Isolation, Durability 的首字母缩写。

### 可移植性与操作的透明性

最后需求发生了变化,考虑一些上面列举的因素的应用,也许需要考虑更多的因素——甚至更糟,需要考虑一组不同的因素。已经存在若干个在不同的方案之间提供统一接口的尝试;比如,DBI和ODBC就是这样的两种成果,它们在相互竞争的关系数据库实现上形成一套一致的API。下一章,我们将雄心勃勃的创建我们自己的一组模块,来隐藏文件与数据库存储之间不同的API。事实上你越是追求访问上的透明性,对性能存在的影响就越大。

下面,我们将要考察各种持续化数据存储的Perl模块。我们按照边界的限制,把它们划分成以下三类:流式的(无记录边界),面向记录的和面向栅格的(关系数据库)。

## 流式数据

本节我们将会看到三种模块,FreezeThaw、Data::Dumper和Storable。所有这些模块都将Perl的数据结构序列化成ASCII或二进制字符串;实际上只有Storable把它们写入磁盘。另两个模块也很重要,因为它们可以同其他持续化存储机制,如数据库和DBM文件联合起来使用。所有这些模块都会正确考虑经过bless操作的对象引用和自参照的数据结构,但是对于typeglob、绑定变量或包含指向C数据类型的指针(可以证明是这样)却无能为力。这些模块还不可能理解隐含的关系。例如,如果你使用第八章“面向对象:下面的几步”中讲述的ObjectTemplate方案,那么这样的“对象”从根本上说就是数组索引,而保存到磁盘上后就是一些缺失数据的没有任何意义的数组索引。另一个微妙的错误,就是当你把引用用作散列表的下标时,Perl会将它们转换成字符串(如SCALAR(0xe3f434))。这并不是一个真正的引用,因此在你将该散列表保存到文件中并重新创建时,隐含的指向原先数据结构的引用将不再有效。

教训:嵌套简单的Perl结构处理起来很容易;对于其他别的情况,在将其写入磁盘以前,你有责任把你的应用数据转换成一种包含普通Perl元素的数据结构。

## FreezeThaw

FreezeThaw, 由 Ilya Zakharevich 编写, 是一种纯 Perl 实现的模块 (没有 C 扩展), 它将复杂的数据结构编码成可以打印的 ASCII 字符串。它并不与文件直接打交道, 而留给你来把编码过的字符串发送到普通的文件、DBM 文件或数据库中。下面是一个使用这个模块的例子:

```
use FreezeThaw qw(freeze thaw); # 加载 freeze() 和 thaw()
# 创建一个复杂的数据结构: 包含数组的散列表
$c = { 'even' => [2, 4, 6, 8],
       'odd'  => [1, 3, 5, 7] };
# Create sample object
$obj = bless { 'foo' => 'bar' }, 'Example';
$msg = freeze($c, $obj);
open (F, "> test") || die;
syswrite (F, $msg, length($msg)); # 也可以使用 write() 或 print()
```

freeze() 函数接收一个标量变量列表并返回一个字符串。数组和散列表必须以引用的形式传递。thaw 方法读取一个编码过的字符串并返回同样的标量变量列表:

```
($c, $obj) = thaw($msg);
```

我们将在第十三章“网络计算: RPC 的实现”中使用 FreezeThaw, 通过一个套接字连接发送数据结构。因为编码形式为 ASCII, 因此我们不必担心诸如字节顺序、整数和浮点数的长度等与机器相关的细节问题。

## Data::Dumper

Data::Dumper 由 Gurusamy Sarathy 编写, 与 FreezeThaw 在主旨上是类似的, 但是却采用了另外一种不同的方式。它把标量变量列表转换成打印格式美观的 Perl 代码, 而这些代码可以存储到文件中并可以在以后进行 eval 操作。考虑下面的代码:

```
use Data::Dumper ;
# 创建复杂的数据结构: 包含数组的散列表
$c = { 'even' => [2, 4, ],
       'odd'  => [1, 3, ] };
# 创建样例对象
```



```
$obj = bless {'foo' => 'bar'}, 'Example';
$msg = Dumper($c, $obj);
print $msg;
```

这将会打印出:

```
$VAR1 = {
    even => [
        2,
        4
    ],
    odd => [
        1,
        3
    ]
};
$VAR2 = bless( {
    foo => 'bar'
}, 'Example');
```

`Data::Dumper` 为每一个标量变量分配一个任意的变量名, 如果你接着就要进行 `eval` 操作, 并重建原先的数据的话, 这没有多大用处。该模块还允许你通过使用 `Dump` 方法来设置自己的变量名:

```
$a = 100;
@b = (2,3);
print Data::Dumper->Dump([$a, \@b], ["foo", "*bar"]);
```

打印结果为:

```
$foo = 100;
@bar = (
    2,
    3
);
```

`Dump` 接收两个参数: 一个是要被打印的指向标量变量列表的引用, 另一个是指向相应变量名列表的引用。如果名字前缀有“\*”的话, `Dump` 将输出相应的变量类型。也就是说, 不是赋值给 `$b` 一个指向匿名数组的引用, 而是赋值给一个真正的列表。你可以将 `Dump` 替换成 `Dumpx`, 以利用完成相同功能的 C 扩展来获得四至五倍的速度提升。

Data::Dumper 还允许你指定定制的子例程来完成对数据的序列化和逆序列化操作，这样可以使你克服前面所提到的棘手问题。请参阅文档以获得详细信息。

## Storable

Storable 是一个直接将数据序列化到文件中的 C 扩展模块，它是这三种方案中最快的一种。store 函数以一个指向数据结构（根）的引用和文件名为参数。retrieve 方法完成相反的工作：给定文件名，返回数据结构的根。

```
use Storable;
$a = [100, 200, {'foo' => 'bar'}];
eval {
    store($a, 'test.dat');
};
print "Error writing to file: $@" if $@;
$a = retrieve('test.dat');
```

如果你需要存储到文件中的数据结构不止一个，你只需将它们放到匿名数组中并将这个数组的引用传递给 store。

你可以将一个打开的文件句柄传递给 store\_fd 来代替文件名。函数 nstore 和 nstore\_fd 可以被用来以“网络”的顺序存储数据，这样可以使数据与机器无关。当你使用 retrieve 或 retrieve\_fd 时，数据会自动的转换回本地的机器格式。（在进行存储时，该模块会保存一个标志来指示数据是否以与机器无关的格式进行存储。）

## 面向记录的方案

在这一节，我们将学习三个本质上依赖于 DBM 库的模块。DBM 是一种基于磁盘的散列表库，起初是由 Ken Thompson 为第七版 Unix 系统编写的。这个功能库从那以后就衍生出了许多变种：SDBM（简单 DBM，一个与 Perl 捆绑的公用域模块），NDBM（新 DBM，打包于一些操作系统中），GDBM（来自于自由软件基金会所支持的项目）。所有这些功能库都可以通过等价的 Perl 模块进行存取，它们利用 Perl 的 tie 机制，提供对基于磁盘的表的透明存取。性能与可移植性是从中进行选择的唯一条件。要注意的是，由这些方案所产生的文件是不可以互换的。

## DBM

我们在这里使用 SDBM 模块，因为它与 Perl 捆绑在一起。SDBM\_File 模块为这个扩展提供了一层包裹程序：

```
use Fcntl;
use SDBM_File;
tie (%capital, 'SDBM_File', 'capitals.dat', O_RDWR|O_CREAT, 0666)
    || die $!;
$capital{USA}      = "Washington D.C.";
$capital{Colombia} = "Bogota";
untie %capital;
```

tie 语句把内存中的散列表变量 %capital 与基于磁盘的散列表文件 *capitals.dat* 联系起来。对 %capital 的读写操作会自动地转换成对相应文件的存取。untie 将释放这种关联，并将任何后续的改变刷新到磁盘上。从 Fcntl 导入的“常量”O\_RDWR 和 O\_CREAT 用来指定打开文件 *capitals.dat* 来进行读写并当文件不存在时创建它。本例中文件的模式（存取控制掩码）被设置为 0644 —— 这是 0666 & ~022 所得到的结果，这里 022 是 umask 值。

我们以前提到过，这种 DBM 方案最大的问题就在于，绑定的键-值对中的值必须为字符串或数字；如果是引用的话，这些模块将无法自动的进行间接访问。因此要想使用键值与复杂数据结构的关联，就必须使用 Data::Dumper 或 FreezeThaw 来将数据进行序列化。这也是我们下面就要讲到的 MLDBM 所做的工作。

## MLDBM

Gurusamy Sarathy 的 MLDBM 模块（多层 DBM）可以将复杂的数据结构存储到 DBM 文件中。它使用 Data::Dumper 来完成对任意数据结构的序列化工作，并使用一个由你选择的 DBM 模块（默认使用 SDBM\_File）来将数据写到磁盘中。下面是一个应用的例子：

```
use SDBM_File;
use MLDBM qw (SDBM_File);
use Fcntl;
```

```

tie (%h, 'MLDBM', 'bar', O_CREAT|O_RDWR, 0666) || die $!;
$sample = {'burnt' => 'umber', 'brownian' => 'motion'};
$h{pairs} = $sample;    # 创建基于磁盘的散列表的散列表
untie %h;

```

跟在“MLDBM”后面的所有tie参数，都将简单的传递给在use语句中指定的模块。

## Berkeley DB

DB 也称作 Berkeley DB，是一个公共域中的包含数据库存取方法的C函数库，这包括：B+ 树，扩展线性散列和定长/变长记录。最新的发行版还提供了对并发更新、事务和恢复的支持。与之相应的 Perl 模块 DB\_File 提供了对 B 树和散列实现的 DBM 包裹程序，还为定长/变长记录提供了绑定数组包裹（也称为 recno 存取方法）。

该 DBM 的用法与前面章节提到的相同。tie 语句的用法如下所示：

```

use DB_File;
use Fcntl;    # 用来导入常量 O_RDWR 和 O_CREAT
tie (%h, 'DB_File', $file, O_RDWR|O_CREAT, 0666, $DB_BTREE);

```

常量 \$DB\_BTREE 用来告诉功能库使用 BTREE 格式，它允许键-值对被保存在一个有序的、平衡的多叉树中；也就是说，键值以词典顺序存储。你也可以像下面这样指定定制的排序子例程：

```

$DB_BTREE->{'compare'} = \&sort_ignorecase;
sub sort_ignorecase {
    my ($key1, $key2) = @_;
    $key1 =~ s/\s*/g;    # 去处空白字符
    $key2 =~ s/\s*/g;
    lc($key1) cmp lc($key2);    # 比较时忽略大小写
}

```

现在，当你使用 keys、values 或 each 来从绑定的散列表中获取数据时，你得到的将是按照定制排列顺序的数据。普通的散列表和其他的 DBM 模块不会为你提供这样的机制。

你也可以使用 \$DB\_RECNO 来代替 \$DB\_BTREE，它使用 TIEARRAY 来将文件看作变长记录的集合：

```
use Fcntl;
use DB_File;
tie (@l, 'DB_File', 'foo.txt', O_RDWR|O_CREAT, 0666, $DB_RECNO);
print $l[1];                # 获取第 2 行
$l[3] = 'Three musketeers';  # 修改第 4 行
untie @l;
```

正像我们在第九章“绑定”中所提到的那样，当前的 TIEARRAY 实现只允许使用数组索引；而不支持 push 和 splice 这样的操作符。DB\_File 模块还提供了名为 push、pop、shift、unshift 和 length 等额外的方法，用法如下：

```
$db = tied @l;
$db->push($x);
```

## 关系数据库

关系型数据库已经存在一段时间了，尽管大多数商业实现都提供了标准化的 SQL，但是它们的本地 C API 却大相径庭。已经有了若干解决方案。微软主动推广普及了 ODBC（开放数据库连接），它已经成为 PC（Wintel）世界中事实上的标准，并为大量的关系数据库提供了标准的前端。在 PC 上使用 ActiveWare 移植版本 Perl 的用户，可以通过使用 Win32::ODBC 模块来存取 ODBC 库。

同时在 Perl/Unix 世界中，Tim Bunce 和其他曾经为不同的数据库编写互不兼容包裹模块的开发人员，提出了 DBI（数据库接口）规范与实现，来融合多方的努力。DBI 在主旨和接口上与 ODBC 规范是相似的。

最近 ODBC 规范被接受作为 SQL CLI（调用级接口）ISO 标准的基础，这样，最终所有的数据库生产厂商都会提供与之相适应的客户端功能库。到这种情形成为现实时，可以预期，DBI 的实现将会被重新编写以利用这种接口，否则就会彻底消失。

在这一节，我们将同时来看一下 DBI 和 Win32::ODBC。

## DBI（数据库接口）

在那些模块和动态加载还没有内建在Perl中的日子里，必须连接数据库包裹函数来创建定制的Perl可执行程序，比如sybperl（为Sybase定制），oraperl（为Oracle定制）等等。这些库函数后来被重新编写，以利用Perl5的功能，但老式的API保存了下来，这就意味着你为一个数据库编写的脚本程序，在另一个数据库上将无法工作。如果你需要可移植性的话，DBI模块是唯一的选择。DBI调用称为DBD（数据库驱动程序）的模块，它们都专门针对某一种数据库产品，以驱动内置的厂商API。例如，如果你使用Oracle数据库，你可以使用oraperl来获得更好的数据库性能，或者你使用DBI和DBD:Oracle组合来获得可移植性。Oraperl和DBD:Oracle都是基于相同的底层代码。下面列出的由Alligator Descartes维护的Web站点是一个包含DBI信息的极棒的知识库：<http://www.symbolstone.org/technology/perl/index.html>（译注2）。

DBI使用起来很简单，你只需与相应的数据库建立连接，并发出SQL查询指令就行了（注4）：

```
use DBI;
$dbname = 'empdb'; $user = 'sriram';
$password = 'foobar'; $dbd = 'Oracle';
$dbh = DBI->connect ($dbname, $user, $password, $dbd);
if (!$dbh) {
    print "Error connecting to database; $DBI::errstr\n";
}
```

connect返回一个代表与特定数据库连接的数据库句柄。例子中的参数\$dbd用来告诉它加载DBD::Oracle模块。这个参数后面还可以跟上指向驱动程序散列表的引用或与连接有关的属性。一些数据库产品允许创建多个连接。

所有的DBI语句在失败时返回undef。错误代码和错误信息可以从\$DBI::err和DBI::errstr中获得；它们反映最近执行的一条DBI语句的错误信息。

---

译注2： 由Alligator Descartes和DBI之父Tim Bunce合著的《Programming Perl DBI》一书是DBI这一主题的权威著作。英文版由O'Reilly公司出版。中文版《Perl DBI编程》已由中国电力出版社出版。

注4： 我将假定你熟悉SQL语言。

## 基本 SQL 存取

SQL 语句可以像下面这样来执行（等价于嵌入式 SQL 中的 `execute immediate`）：

```
$dbh->do("delete from emptable where status != 'active'");
print "Error: $DBI::err .... $DBI::errstr" if $DBI::err;
```

如果将同样的查询或与之类似的查询执行多遍，那么系统将会被迫对它进行一遍又一遍的语法分析。为了避免这种负载，你可以使用 `prepare` 来编译一个参数化的查询，并使用 `execute` 来进行多遍执行。

`prepare` 方法将接收以 “?” 表示的参数位置的查询：

```
$sth = $dbh->prepare ('insert into emptable (id, name, age)
                      values (?, ?, ?)');
```

你可以使用返回的语句句柄来不停的执行这条语句，每次为相应的参数位置提供一个值数组。这些值有时也被称做联编参数。实际上，`do` 在内部也准备和执行给定的查询。

下面的代码从标准输入中读取雇员的名字和年龄，并使用上面创建的语句句柄将行插入到数据库中：

```
while (defined($line = <>)) {
    chomp($line);

    # id, name, age 用制表符分开
    ($id, $name, $age) = split (/\\t/, $line);
    $sth->execute($id, $name, $age);
    die "Error: $DBI::err .... $DBI::errstr" if $DBI::err;
}
```

如果某个字段可以为空，你可以给 `execute` 传递 `undef` 来表示空值。

## Select

下面的例子描述了如何使用 SQL `select` 语句来获取大量的信息：

```
$cur = $dbh->prepare('select name, age from emptable where age < 40');
$cur->execute();
die "Prepare error: $DBI::err .... $DBI::errstr" if $DBI::err;
while (($name, $age) = $cur->fetchrow) {
    print "Name:$name, Age: $age \n";
}
$cur->finish();
```

和前面一样, `prepare` 语句返回一个语句句柄。在执行时, 这个句柄在内部与一个数据库游标相关联, 并被用来获取数据库返回的每一行信息。 `fetchrow` 返回与 `select` 查询中指定字段所对应的值。 `finish` 关闭游标。

### 查询元数据

一旦一条语句准备好并予以执行, DBI 将会以语句句柄属性的形式来保存下面的信息:

`$DBI::rows`

作用或返回的行数

`$sth->{NUM_FIELDS}`

由 `select` 语句返回的字段数

`$sth->{NUM_PARAMS}`

由任意查询返回的参数个数

在执行完一条 `select` 语句后, 下列属性中包含有指向特定字段信息数组的引用:

`$sth->{NAME}`

由查询所返回的列名称

`$sth->{NULLABLE}`

表示字段是否可以为空的布尔值

`$sth->{TYPE}`

字段类型

`$sth->{PRECISION}`

字段的浮点数精度



```
$sth->{SCALE}
```

字段长度

让我们来应用一下到目前为止所学的东西,来创建一个大多数关系数据库都配备的交互式 SQL 前端(如 sqlplus 和 isql)的替代产品。如例 10-1 所示:

例 10-1: sql.pl:交互式 SQL 前端

```
use DBI;
$dbname = 'DEMO732'; $user = 'scott';
$password = 'tiger'; $dbd = 'Oracle';

$dbh = DBI->connect($dbname,$user,$password,$dbd) ||
    die "Error connecting $DBI::errstr\n";

while(1) {
    print "SQL> ";                # 提示符
    $stmt = <STDIN>;
    last unless defined($stmt);
    last if ($stmt =~ /^s*exit/);
    chomp ($stmt);
    $stmt =~ s/;/\s*$/;

    $sth = $dbh->prepare($stmt);
    if ($DBI::err) {
        print STDERR "$DBI::errstr\n";
        next;
    }
    $sth->execute();
    if ($DBI::err) {
        print STDERR "$DBI::errstr\n";
        next;
    }
    if ($stmt =~ /^s*select/i) {
        my $rl_names = $sth->{NAME};                # 列名数组的引用
        while (@results = $sth->fetchrow) {         # 检索结果
            if ($DBI::err) {
                print STDERR "$DBI::errstr,\n";
                last;
            }
            foreach $field_name (@$rl_names) {
                printf "%10s: %s\n", $field_name, shift @results;
            }
        }
    }
}
```

```
        print "\n";
    }
    $sth->finish;
}
$dbh->commit;
```

这个脚本程序准备并执行所有的语句。如果语句为 `select` 查询，它将获取每一行数据，并打印出注有相应列名的字段值。注意，`fetchrow` 在标量变量环境中，将返回指向值数组的引用。

## 事务

当使用 `connect` 创建数据库连接成功时，DBI（或数据库）会自动启动一个新的事务。你可以在数据库句柄上使用 `commit` 或 `rollback` 方法来结束一个事务；这时将会隐含的启动一个新的事务。这里 XA 标准定义的分布式事务并不支持。

## 特殊函数

我们可以在数据库句柄上使用 `func` 方法来调用特定于驱动程序的函数。例如，mSQL 数据库驱动程序提供了一个名为 `_ListFields` 的内部函数，它返回表中有关列的信息。我们如下所示来调用它：

```
$ref = $dbh->func($table, '_ListFields');
```

显然，使用 `func` 不是一种可移植的方案。

## DBI 没有提供的功能

列出 DBI 当前没有提供接口的常见数据库任务是有益的。这里并不是藐视 DBI/DBD 的实现者。而是表明这样一个事实，数据库在标准化委员会所没有涉及方面，存在着极大的多样性。

## 元数据

DBI 提供了方法 `$dbh->tables()` 来获取所有可以进行存取的表名的列

表。然而并不存在返回表中列名的函数。幸运的是，存在一种简单而具可移植性的解决方案。由于`select`查询返回元信息，我们可以使用这样一个“哑”询问，我们确信它不会返回任何记录，但又能够成功执行：

```
select * from $table where 1 = 0;
```

这里的`where`子句完全有效，但是这个条件永远不会满足。“\*”使它返回所有的列。这样我们就可以像在“询问元数据”一节中所讲的那样，来考察`$sth`的属性获取所需要的信息。

### 创建数据库

在如何创建上，数据库API差别相当的大；你必须使用特定产品的API来完成这种任务。一旦数据库被建立起来，就可以使用DBI来创建或删除其中的表了。

### 从数组中插入或创建

从数组中大批量的插入和更新数据并不是标准SQL CLI（注5）的功能。如果确实有大量数据需要进行插入的话，你最好将数据卸载到文件中，并使用相应的大批量拷贝工具（如Sybase的`bcp`）将其高速传送到数据库中。（为了得到更好的性能，你可以在加载数据前先将索引删除并在以后再予以重建。）

### 存储过程和触发器

存储过程和触发器对于不同的产品差别非常之大。所有的Perl数据库模块如`oraperl`和`sybperl`，都提供对它们本地数据库功能的存取，但是DBI并没有试图来为此提供一致的接口。请你参照模块文档来查看详细信息或访问DBI的网站来了解建议的方案。

### 统一的错误代码

DBI或许是可移植的，但是它并没有提供一套可移植的常用错误代码。例如，假定你在其不存在的情况下创建一个表。你也许会采用下面的方法：

```
$dbh->do("create table emptable (id char(15), name char(40),  
                                         age integer)");
```

---

注5： 调用层接口（Call Level Interface）——所有数据库厂商都会支持的标准C API的另一种称呼。

如果 `$DBI::err` 中包含有错误代码,若是它表示诸如“表/视图已经存在”等内容你不会把它当回事。不幸的是,如果你在使用 Oracle 数据库,这个错误代码将是 955,而对于 Sybase 来说则又是另一个完全不同的数。这种可移植性是不行的!

## Win32::ODBC

Win32::ODBC 模块在 ActiveWare 移植到微软 Windows 操作系统上的版本中可用,它与 DBI 的方法类似。考虑下面的脚本程序,它从雇员表中提取所有的记录:

```
use Win32::ODBC;
$dbh = new Win32::ODBC ($dbname);
if $dbh->Sql("select * from emptable") {
    print 'Error: ', $dbh->Error(), "\n";
    $dbh->Close();
    exit(1);
}
@names = $dbh->FieldNames();
while ($dbh->FetchFrow()) {
    # Data returns the values
    @values = $dbh->Data();
    ... do something with @names and @values.
}
```

这里的 `Sql` 语句等价于 DBI 中的 `do`。ODBC 没有语句句柄的概念;而是使用数据库句柄来获取上一条查询结果。

有两种方法来得到元数据: `TableList`, 它将返回表名的列表, `ColAttributes`, 它将返回当前记录中所提供的每个字段名的指定的属性。

## 相关资源

1. Transaction Processing: Concepts and Techniques. Jim Gray and Andreas Reuter. Morgan Kaufman, 1992.

现有的知识最为丰富和可读性强的一本资料,如果有什么有关事务的内容没有在这里描述的话,那它或许就不存在!

2. An Introduction to Database Systems, Volumes I and II. C.J. Date. Addison-Wesley, 1994。

透彻描述了有关持续性的问题和数据库技术。

3. Berkeley DB library。

参见 <http://mongoose.bostic.com/db/>。

4. 由 Alligator Descartes 维护的 DBI 站点: <http://www.hermetica.com/technologia/DBI/>。



# 第十一章

## 对象持续性 的实现

本章简介:

- 适配器介绍
- 设计注意事项
- 实现

上帝赋予我们记忆，这样即便是在11月份  
我们也会拥有玫瑰

——James Matthew Barrie

亚马逊河由两条河流汇集而成：一条是夹杂泥沙的黄色的索利蒙伊斯河，另一条是戏剧性的拥有黑而发亮的河水的内格罗河（注1）。在它们汇合后顺流而下的12英里当中，虽然分享着相同的河床，但却顽固保持着各自鲜明的特色。这似乎多少与我们正在讲述的专题——对象的持续性有着一种奇特的相似之处。

在商业计算世界里有两个重要的阵营：面向对象的供给者（语言设计者，对象技术的倡导者）和持续性的供应商（数据库和事务处理监控器的实现者）。像索利蒙伊斯河与内格罗河那样，这两个阵营（和拥有不同侧重的多个阵营）都有着它们各自的计划，即便它们会在将来的某个时候尽力进行融合。

对面向对象技术人员来说，没有比商业级的持续性存储更令人倾心的东西了，他们提出方法来改进各种各样的持续性存储技术以达成一种对象模式。其中一些杰出的成果包括由开放管理组织（OMG）提出的CORBA持续性服务规范，Sun的PJava（持续性Java）和微软公司的OLE持续性框架。与此同时，数据库技术人员则为他们产品嫁接面向对象的特性：诸如Informix和Oracle等数据库生产厂商已经宣布了对对象关系型数据库（支持抽象数据类型，而不仅仅是普通的标量变量），还有Tandem、IBM、Tuxedo和Encina等公司的事务处理监控器产品也宣

注1： 这种颜色源于沉积的矿物质和沿岸沼泽中分解的有机物。

称提供面向对象的接口。存在一个很小的由面向对象数据库厂商组成的称作“对象数据库管理组 (ODMG)”的对象持续性阵营,但是它们的影响几乎不值一提(从商业上来说)。

这些组织中讨论的热点话题之一,就是“互不相关的”持续性,也就是指可以使应用或对象具有持续性,而无须在对象中嵌入任何或较多的与持续性相关的代码的能力。这个想法非常诱人,这样你就可以设计你的对象模型,在内存中实现它,然后在“另一侧”添加上持续性的功能。使用这种方式,对象就无须操心大量的数据库细节(和差异),也无须处理系统错误、数据的格式等其他的问题(注2)。你可以这样来思考一下:如果你决不会在对象中嵌入与用户界面相关的代码,那么你要持续性做什么?

要获得上面所提到的透明性,传统上有两种解决方案。

一个就是利用系统方面的能力,如硬件、操作系统和编译器。比如,像Object Store和Texas Persistence Store(一个公共域库)这样的面向对象数据库,使用Unix的系统调用mmap和mprotect来透明地将数据在磁盘和内存间进行转存。另一个有意义的面向系统的方案出自贝尔实验室的一个小组,他们创建了一个功能库来有控制的将应用系统的状态进行核心转存,这样就可靠的将所有基于内存的数据结构保存到磁盘上(注3)。他们还为此种机制增加了恢复与事务功能,这样差不多可以保证对应用系统完全透明。

第二种获得透明得或互不相关持续性的方案,是提供应用层的工具和功能库,这种方案要比面向系统的方案的可移植性好得多。例如,CASE工具生成代码自动地完成把对象发送到持续存储(通常是一个数据库)中的工作。而MFC(微软基本类库)这样的技术则要求对象将自己转换成数据流并写入到文件中。对于后面这种情况,对象必须实现流方法。任何一种情况下需要手工编写的代码都相当少,因此这还算是一种相当透明的方案。

---

注2: 如果需要一部有关这个主题的精彩文献的话,就请参阅PJava设计论文(见本章后的“相在资源”[4])。

注3: 注意,Perl的dump操作并不产生core文件,但是它会终止应用,这是一项不怎么令人愉快的功能。

这一章我们将要讨论一种作做适配器 (Adaptor) 的试验性项目, 它为 Perl 对象提供持续性框架 (它当然是用 Perl 来编写的)。这是一种应用层的解决方案, 而且不要求对象实现与持续性有关的方法。不像典型的 CASE 工具, 它不产生任何代码文件, 这是因为 Perl 是一种动态语言。

适配器项目的首要目的就是为了学习互不相关的持续性; 我想, 它可以通过使用完全独立于对象的信息, 将对象调整为能适应特定类型的方法来完成。本章中所描述的实现要依赖配置文件来确定, 哪些属性映射到数据库的哪些列以及如何映射等信息。

该项目的第二个目的是, 你在获得查询和事务原子性操作的情况下, 会如何以不同的方式来编写应用。也就是说, 即便是你根本就没有数据库, 假设你请求某个实体, 如“提供给我所有工资超过\$100000的雇员”, 而应用从一开始就已经是持续性就位了。我坚信你不可能凭空给应用添加持续性; 在知道有某种持续性能力存在的情况下 (即便它们还不十分清楚持续性的类型), 对象的实现将看起来非常不同。这与在知道它将来可能拥有图形用户界面, 并且是事件驱动的情况下应用很相似。例如你不能再将错误信息输出到 STDERR, 而且还要保证没有代码会在 I/O 时无限制的阻塞。(实际上, 我们要在第十四章“使用 Tk 进行用户界面编程”中讨论这些问题。)

本章中所引出的问题, 或许要比实现的具体细节更为重要; 然而要清楚的理解这个问题, 一种实现是必须的。

## 适配器介绍

如图 11-1 所示, 适配器打算作为一组模块, 将一种一致的持续性接口转换成针对特定类型的持续性存储接口的模块。本章将描述两个模块的实现: `Adaptor::File`, 它能够将对对象存储到普通文件中; `Adaptor::DBI`, 它则能够将对对象存储到关系数据库中。从现在起, 我们将使用“适配器 (adaptor)”一词来表示任一模块中的对象。

适配器代表一种典型的持续性存储, 它能够容纳一组异构的对象; `Adaptor::File`



对象是对文件的包裹层，而 Adaptor::DBI 是对数据库连接的包裹层。所有的适配器都提供对基本 SQL（注 4）查询和事务的支持（注 5）。

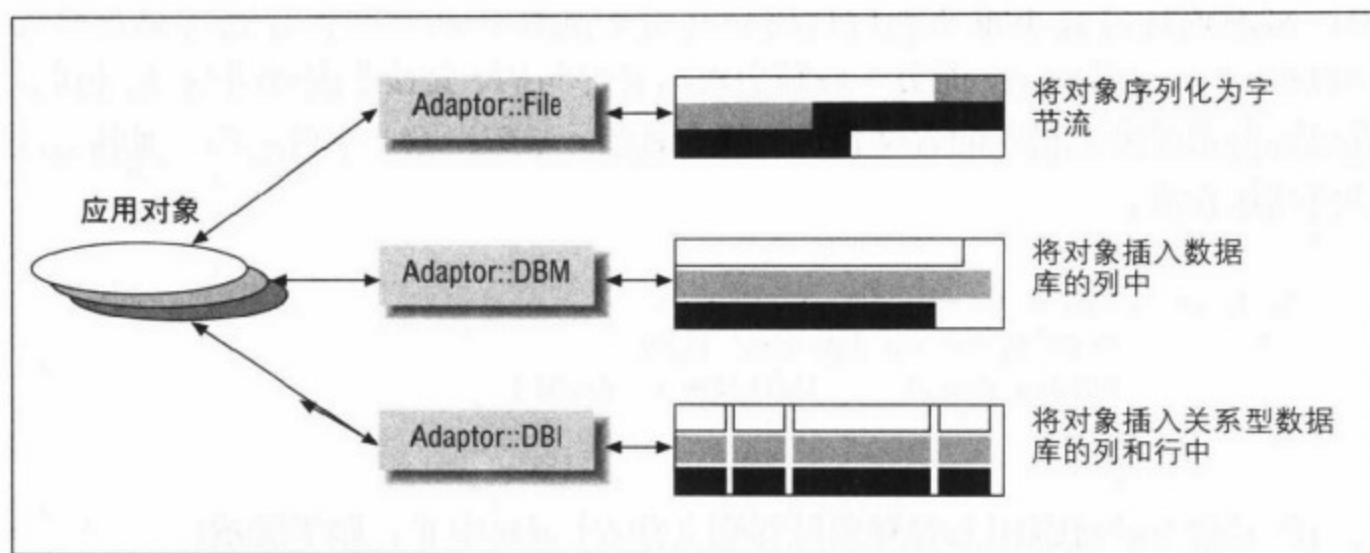


图 11-1 适配器模块

在我们使用这些模块之前，让我们来创建几个测试用的应用对象。我们将使用第八章“面向对象：下面的几步”中讨论的 ObjectTemplate 库：

```
use ObjectTemplate;
#-----
package Employee;
@ISA = ('ObjectTemplate');
attributes qw(_id name age dept);
#-----
package Department;
@ISA = ('ObjectTemplate');
attributes qw(_id name address);
#-----
$dept = new Department (name => 'Materials Handling');
$emp1 = new Employee   (name => 'John', age => 23, dept => $dept);
$emp2 = new Employee   (name => 'Larry', age => 45, dept => $dept);
1
```

我们现在有了三个对象，不带任何与数据库有关的代码。为了把这些对象存储到一个持续性存储中，我们先来创建一个文件或数据库的实例，如下所示：

注 4： 只有 SQL 的 where 子句，而不是整个 select 子句；也不支持连接。

注 5： Adaptor::File 实现了一种相当简单的模型，但是它确实支持这种接口。

```
$db = Adaptor::File->new('test.dat', 'empfile.cfg');
```

适配器对象 `$db` 现在与文件 `test.dat` 相关联并将所有提供给它的对象保存到文件中。对象或许会有不希望进行持续性存储的属性，如一些属性是计算得到的 (`after_tax_salary`)，而另一些则会指向文件句柄、套接字或 GUI 组件。因此，适配器要求开发人员在配置文件（本例中是指 `empfile.cfg`）中指定哪些属性要进行持续性存储。

```
%g_attr_names = (
    "Employee" => [qw(name age)]
    "Department" => [qw(name address)]
);
```

该适配器现在就可以用来将对象保存到文件 `test.dat` 中了，如下所示：

```
$db->store($dept);
$db->store($emp1);
$db->store($emp2);
```

我们的“数据库”中现在已经有了好几个对象，我们于是就可以使用 `retrieve_where` 方法来查询这个数据库，如下所示：

```
@emps = $db->retrieve_where ('Employee', "age < 40 && name != 'John'");
foreach $emp (@emps) {
    $emp->print();
}
```

这个方法以一个类名和查询表达式为参数，并返回匹配条件的指定类的对象引用。

`flush` 方法被用来确保内存中的数据刷新到磁盘上：

```
$db->flush();
```

你也可以将对象作为事务来存储：

```
$db->begin_transaction();
$db->store($emp1);
$db->store($emp2);
$db->commit_transaction(); # 或 rollback_transaction
```

文件适配器保留所有传递给 `store` 方法的对象，并在执行 `commit_transaction` 时，将它们刷新到磁盘上。否则如果你调用 `rollback_transaction`，它将忽略所有内部数据结构，并从文件中重新加载，以消除你对对象所做的任何修改。这决不是一种真正的事务（它并不提供系统失效时对数据的保护），但它确实提供原子形式的更新，可以用做一种自动回退机制。

要想将这些对象存储到数据库而不是文件中，我们只需使 `$db` 成为 `Adaptor::DBI` 的一个实例即可。别的一切都无须改变，除非你想使用真实的事务机制来保证数据的安全才安心。

`Adaptor::DBI` 的构造函数参数是数据库相关的：

```
$db = Adaptor::DBI->new($user, $password, 'Sybase', 'empdb.cfg');
```

这个方法将使用前三个参数来调用 `DBI::new` 方法。最后一个参数与前面一样是一个的配置文件，但额外还包括与数据库相关的映射信息。

```
[Employee]
table      = emp
attributes = _id, name, age
columns    = _id, name, age
[Department]
table      = dept
attributes = _id, name, address
columns    = _id, name, address
```

参数 *attributes* 指定了要从给定模块实例中提取的属性的列表，*columns* 则列出了数据库中与之对应的列的名字。许多适配器可以使用相同的配置文件。

## 设计注意事项

适配器接口无疑很简单，在这一节，我们将会问自己这是否是太简单了。该适配器的实现还处在原型阶段，但是我们下面将会看到，它现在已经足以应付那些需要对象持续性的人们所要应付的所有问题。

## 设计目标

我想使适配器 API 具有透明性，也就是说可以任意的改变持续性存储的类型。这种想法可以使我们不必涉足数据库就能编写小的系统原型，继而只需改变适配器就可以迁移到真实环境的数据库中。进一步，我还想保留这种对象可同时存储在多个持续性存储中的灵活性，因为这是把一个对象从一个存储中拷贝到另一个中去的唯一方式。

我想尽可能的保留那些基于内存的数据结构（易于定位，速度，易用性）和基于数据库的数据结构（事务，并发存取，查询）的最佳特性。最后，我不想适配器破坏对象的封装性，也就是说这种实现不能对一个模块如何来存储实例相关信息和如何构造对象作任何假设。

## 对象封装

一个我们很容易忘记的重要限制就是对象不仅仅是数据。我们在上一章所见到的那三个序列化模块——FreezeThaw、Data::Dumper 和 Storable 对做了这样的假定。它们检查对象引用下面所隐藏的数据结构并将它们所能达到一切序列化。这样假定了所有实例相关的数据都可以通过引用来达到：这是一个错误的假设。例如，一个 ObjectTemplate 类型的对象引用只是一个对标量变量的引用。通过考察那个引用，你无法知道对象的属性。

对于上面提到的这些模块还有更糟糕的问题：当从字节流中恢复对象时，它们只是在内存中重新创建原来的数据结构并将其在目标模块中进行 bless，而无须模块的参与。这样有可能缺少几项关键的初始化工作。

为了防止这些问题的发生，适配器要求需要持续性的每个类都支持三种方法：构造函数，new() 和两个属性存取方法 get\_attributes() 与 set\_attributes()，下面将详细讲解：

1. new(): 模块必须提供构造函数（相当于 C++ 中的“默认构造函数”），它能够在没有任何输入参数的条件下创建一个对象。一种用于创建基于散列表对象的最简单的默认构造函数如下所示：

```
sub new {  
    bless {}; # bless 一个散列表引用并将其返回  
}
```

当然，另一个更简单的方式是使用 `ObjectTemplate`，它提供了一个可继承的默认构造函数。而且，它还提供了下面要列出的两个方法。

2. `get_attributes(LIST)`: 提供一个包含属性名的列表，这个方法将会返回与之对应的值的列表。现在暂时的限制是这些值必须为标量变量（一个非常大的限制；我们马上就要更多的谈到这一点）。因为这个方法编码高效，对调用单一存取函数的适配器来说是可取的。例如，如果你使用一个散列表来存储对象，你可以将该方法实现为一个散列表切片：

```
sub get_attributes {  
    my $obj = shift; # 现在 @_ 中包含了属性名  
    @{$obj}{@_};     # 散列表切片返回对应的值  
}
```

适配器使用配置文件来指定持续性属性的列表。

3. `set_attributes(LIST)`: 给定属性名值对的列表，该方法将更新相应的属性。这个函数和上面提到的 `get_attributes` 函数必须允许一个名为 `_id` 的属性，这样做的原因我们马上就要谈到。

这些方法完全是些通用的函数；它们与持续性没有任何程度上的依赖性。作为对比，一些功能库，尤其是在 C++ 世界中（微软的 MFC 类和 NIH 库）要求对象支持一个流界面。由于流式对象对数据库来说毫无用途，我选择有区别的保存属性。况且，如果我们想要将这些属性发送到文件中，我们知道总是可以依赖其他的模块来将它们流化而无须让对象来完成这样的工作。

## 对象 - 适配器协议

在进行对象存储时，适配器参阅配置文件以获得这个类进行持续性存储所需的属性列表。它把这个列表传递给 `get_attributes` 来获取相应的值，然后根据适配器的类型，或者将其序列化到文件中或是使用 SQL 查询来更新数据库。

在从数据库中获取对象时，适配器将调用相应类的 `new()` 方法，然后调用 `set_attributes` 并使用从持续性存储中获得的数据来初始化新创建的对象。

## 多值属性与数据库映射

Adaptor::DBI只是简单的把对象翻译成关系数据库表中单一的一行。因此，它要求由 `get_attributes` 返回的每个值必须是一个标量变量（数字或字符串，而不是引用）。我希望最终通过使用类型映射（*typemap*）来取消这种限制。类型映射是由几组代码来完成对数据类型进行定制转换（注6）。

下面是一些当前可用的如何处理包含一个或多个非标量变量类型属性对象的方法：

1. **定制的 {get,set}\_attributes:** Adaptor::DBI允许内存中的多值属性。它所要求的只是 `get_attributes` 将这些属性转换成将来 `set_attributes` 能理解的标量变量形式。这样，当数据被从磁盘上读回时，`set_attributes` 将能够把它转换回原先的结构形式。它可以使用 `FreezeThaw`、`Data::Dumper`、`sprintf` 或 `pack` 来完成这种转换工作。后两种方法或许是最好的，因为你可以控制最终产生的标量变量的长度（这很要紧，因为数据库列的大小是预先定义好的）。该标量变量可以被映射到能容纳变长数量字符的（如 `VARCHAR`）或二进制字符串的数据库列中（如 Oracle 的 `RAW` 或 `LONG RAW`）。附带说一下，`BLOB`（二进制大对象）列仍有许多问题，如一些数据库只允许一个 `BLOB` 列，而其他的却提供一种与常规数据类型截然不同的 API。
2. **使用文件存储:** Adaptor::File 并不关心属性是引用还是普通的标量变量，因为它只是简单的把它们传递给 `Storable`。换句话说，如果你使用的是 Adaptor::File，那么 {get/set}\_attributes 就无须担心多值属性。当然如果你明天就决定使用数据库的话，这个方法可不行。还有一种危险性就是你可能无意识的存储了毫不相关的对象，而这仅仅是因为它们碰巧在一些属性中被用到。
3. **分离的对象类:** 如果一个属性是一系列同构记录的引用（如一个雇员可以拥有多条描述学历的记录），那么这个属性可以被制定为一个拥有自己单独表项的分离的类。我们将在这一节后面学习对象的关联时，更多的谈到这一问题。

---

注6：我们在第十八章“扩展 Perl：第一课”中将看到，在创建扩展模块中是如何应用类型映射的概念的。

由于`{get, set}_attributes`是通用的方法, 那么它们怎么会知道是否要序列化复杂的属性呢? 哦, 它们无须知道这些。如果你想加以区分的话, 你可以专为持续性来保留另一组属性名 (如 `db_address`), 并让这些方法识别出这些特殊的属性。这个策略有悖于我们起先不在对象中嵌入与数据库有关的代码的打算。那么好了, 正如 Jiri Soukup 在他的书《Taming C++: Pattern Classes and Persistence for Large Projects》中所指出的那样: “现在流行显示雅致的 C++ 程序, 然而雅致这东西不是提供持续性数据存储这类程序的特性。”

## 继承与数据库映射

将一种继承关系映射到数据库的常见策略, 是让超类和导出类分别映射到自己的表中。表示导出类的表中包含了其所有超类的所有属性; 换句话说就是继承层次被展平了。另一种不常用的策略, 就是创建一个包含一个继承层次的所有属性的联合, 并让那个层次的所有类的所有对象使用一张表。你可以使用一个额外的列来标识对象特定的类。适配器对于这两种策略来说都不成问题, 因为它将对属性名和值进行解释的负担全部交给了 `get/set` 方法。

## 对象标识

面向对象世界中的一个关键的理解就是对象的属性和标识是分开的。两个对象可以拥有相同的属性但仍占据着不同的地址空间; 它们可以被认为是相等的而不是相同的。

在内存中, 一个对象的地址提供了它的标识, 而在数据库中, 主关键字完成同样的工作。适配器要求每个对象都支持一个称作 `_id` 的属性, 因此将来的实现可以自动的把关系属性 (那些指向其他对象的属性) 转换成另一端对象的 `_id` 值。例如, 如果你要得到一个雇员对象的 `dept` 属性, 它会向它所指向的部门对象要求其 `_id` 并予以返回。要注意, 这个对象不必为它的 `_id` 分配内存空间; `get/set_attributes` 方法可以根据其他的一些属性来凭空计算出来它的值。比如一个雇员对象可以在要求其 `_id` 时返回其社会福利号或雇员号。

当 `store()` 被调用时, 适配器在它还没有标识的情况下, 将会为对象提供一个唯一的标识。这个标识不能是个简单的全局计数器, 因为当程序重起时, 它将会被



复位为0，于是适配器就会开始分发在前一阶段已经分配给持续对象的数字。将计数器的最新值保存到文件中是一种缓慢的操作，因为你必须在每次存储对象时都要将这个值刷新到文件中。（因为你不知道程序何时会崩溃。）我们当前的实现试验了另外一种方案。在程序开始执行时，它将记下当前时间（使用`time`，它会返回自1970/1/1所流失的秒数），并把它添加到一个五位计数器的后面，这个组合而成的数字可以作为对象标识。当计数器溢出时，时间又被重新记录下来。如果程序发生了崩溃了又再次执行时，标识将是唯一的，除非它崩溃后在一秒中内又重新恢复执行。这种模式的毛病在于它产生的是长标识（使用`pack()`后将是八个字节）。而且它无法工作在分布式的环境中，因为完全存在两个程序在同一时刻调用`time()`而产生相同的标识的可能性。为了防止这种问题的发生，你必须创建一个更大的包含机器IP地址的标识。

## 对象关联

当把对象存储到文件或数据库中时，一个包含指向其他对象引用的属性可以被转换成那个对象的`_id`值（用数据库的术语来说就是外键）。就当前的实现而言，适配器并不能自动的做这种转换，因为我还没有处理下面这种问题的一个好的方案。

假设一个雇员对象的`dept`属性指向一个部门对象。在存储`dept`时，我们简单存储了部门对象的`_id`。到目前为止还没有出现问题。现在，当我们把雇员对象从磁盘中取回时，我们如何来处理这个编码的`dept`属性呢？我们需要马上创建一个内存`dept`属性可以访问得到部门对象吗？如果创建的话，它应该包含怎样的数据呢？我们应该从数据库中读取数据来正确的初始化这个对象吗？这就会出现一个问题，一个正常的查询将会导致从数据库中加载各种类型的对象。或者我们是否应当将部门对象置于一种未初始化的状态，并在它第一次被使用时才初始化呢？进一步说，我们必须确定在从磁盘上读取部门对象的数据时，它不是创建一个新的对象，因为拥有同样标识的对象已经存在于内存中了。我们将在下一节中更多的讲到这个问题。现在暂时让对象来实现外关键字属性来自己轻松一些。

现在让我们考虑一下各种成分之间的关联在数据库中是如何实现的，而不考虑它们在内存中的情况。

一对多的关联，例如一个部门包含一组雇员，可以在多的那一方，以外关键字属



性的方式来实现。也就是说，在数据库中，雇员对象回指包含它的部门对象，而不是让部门来维护一个多值属性。

多对多的关联可以被抽象为一个分离的类；这样，每个关联均变成数据库中的一条记录。例如，一个雇员可以在多个项目下工作；而一个项目又有许多为其工作的雇员。我们可以把这种关系抽象为一个名为 ProjectEmployee 的分离的类。这种模式附加的好处是这种关系可以被查询和更新，而独立于它们所连接的对象。这种关联的势高于二（例如三重关联）将映射到不同的表中。Rumbaugh 等的著作《Object-Oriented Modeling and Design》对数据库的映射方案有精彩的论述。

一旦对象关系数据库广泛应用之后，所有这些策略（或限制）都将会发生戏剧性的改变。

## 对象在内存中的唯一性

在结束了对象标识问题的讨论之后，接着又有一个非常棘手的问题。请考虑下面的查询：

```
@emps = $db->retrieve_where ('Employee', 'age < 40');
```

这会返回符合查询条件的对象引用的列表。现在如果你重新发出这个查询，无非是期望返回相同的对象列表（也就是相同的对象引用）。这意味着适配器必须在内存的高速缓存中保留一份在前一次查询中已经从磁盘中获取的对象，这样如果重读数据库中的一行信息，相应的对象将被重用。这种模式的问题在于，如果该高速缓存位于脚本程序空间中，它会增加所有组成对象的引用记数，也就意味着一旦对象处于高速缓存之中，它将永远不会被释放，即便是没有任何对其感兴趣的对象。换句话说，高速缓存永远不会缩减并且在最坏的情况下，它会包含数据库中的所有对象。

这个问题的一个解决方案，就是用 C 来实现高速缓存，而且压根儿就不更新引用记数（注 7）。如果所有的持续性对象都继承自一个名为 Persistent 的模块的话，那么可以使用该模块的 DESTROY 方法来删除高速缓存中不再需要的对象。

---

注 7： 在阅读了第二十章“Perl 的内部工作”之后，您就会知道如何做到这一点了。

当前实现的 `Adaptor::DBI` 模块拥有一种简单的解决方案，它为每一个查询创建一组对象，并交由 Perl 在没有其他对象引用的情况下自动的将其释放。这就意味着应用开发人员在修改由查询返回的对象时必须十分小心。我知道这是一种笨拙的解决方案。而且现在还没有考虑高速缓存不同步的情况——就是在有人修改了数据库的内容后，高速缓存数据已经过期的问题。

`Adaptor::File` 模块不会出现这个问题，因为它维护着一个提供给 `store()` 方法的所有对象的列表（下一节我们将会讲到这样做的原因）；因此，接下来的相同查询将返回相同的列表。

## 查询

面向对象数据库还不受欢迎的一个重要原因，就是缺乏一种查询语言（或至少是一种标准的查询语言）。如果你在数据库中拥有一百万个对象，那么将每个对象加载到内存中，检查其是否匹配你的条件是一件可怕的事；这件工作最好由数据库来完成。`Adaptor::DBI` 简单的将查询转换成等价的 SQL 查询，而 `Adaptor::File` 则为基于文件的对象实现了一种纯朴的模式：它将查询表达式转换成可计算的 Perl 表达式并对所有的对象进行迭代，检查它们与查询规则是否匹配。

## 模式演变

比如说，你将对象发送到文件中，而且明天要为对象实现增加一些属性。我们称这种情况为模式发生了演变。该应用框架必须能够使老数据与新的对象实现协同一致起来。

## 实现

这一节将讲述 `Adaptor::DBI` 与 `Adaptor::File` 的实现问题。我们将只讲解进行查询处理和基于文件或数据库的 I/O 等关键程序。在你研读代码时要更多的注意设计思想和未实现的功能。

## Adaptor::File

一个 `Adaptor::File` 实例代表存储在文件中的所有对象。在该适配器创建时（使用 `new`），它会读取整个文件并将数据翻译成内存中的对象。将整个文件一下读入内存中避免了对文件中变长数据的随即存取而实现复杂磁盘存储模式；毕竟这是 DBM 和数据库的实现工作。因此，这种方案对于大批量的对象（如超过 1000 个）来说不推荐使用。

该文件适配器有一个名为 `all_instances` 的属性，这是一个包含提供给 `store` 方法的所有对象的散列表（以它们的 `_id` 为索引），如图 11-2 所示。

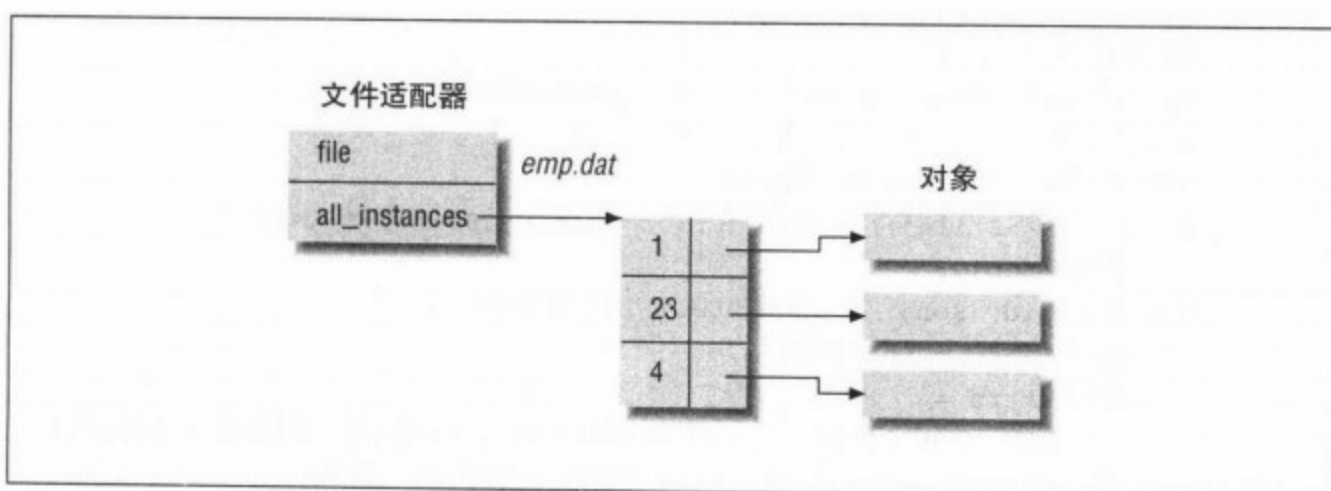


图 11-2 文件适配器的结构

## 存储对象

让我们来查看一下两个用于将对象存储到文件的方法：`store()`和`flush()`。

`store`为对象新分配一个唯一的标识（如果有必要的话）并简单的将对象安插到 `all_instances` 散列表中。它并不把数据发送到磁盘上。

```

sub store {
    # adaptor->store($obj)
    (@_ == 2) || die 'Usage adaptor->store ($obj_to_store)';
    my ($this, $obj_to_store) = @_; # $这里是指 'all_instances'
    my ($id) = $obj_to_store->get_attributes('_id');
    my $all_instances = $this->{all_instances};
    if (!defined ($id)) {
        # 以前没有见到过这个对象，产生一个 id（不必在意这个 id 是如何产生的）
    }
}
  
```

```

        $id = $this->_get_next_id();
        $obj_to_store->set_attributes('_id'=> $id);
    }
    $all_instances->{$id} = $obj_to_store;
    $id;          # 返回对象标识
}

```

注意这里对象被告知它的新标识(使用 `set_attributes`)，因此如果它再次提供给 `store` 的话并不会分配新的标识。

实际把数据写入文件是由 `flush` 来完成的：

```

sub flush {      # adaptor->flush();
    my $this = $_[0];
    my $all_instances = $this->('all_instances');
    my $file          = $this->('file');
    return unless defined $file;
    open (F, ">$file") || die "Error opening $file: $!\n";
    my ($id, $obj);
    while (($id, $obj) = each %$all_instances) {
        my $class = ref($obj);
        my @attrs =
            $obj->get_attributes(@($this->get_attrs_for_class($class)));
        Storeable::store_fd([$class, $id, @attrs], \*F);
    }
    close(F);
}

```

`flush` 只是简单的遍历 `all_instances` 散列表，对每一个用户定义的对象，调用方法 `get_attributes.get_attrs_for_class` 返回每个类的持续性属性列表(以引用数组的形式返回)，这是通过加载提供给适配器构造函数的配置文件来获得的。

这些属性值与类和实例标识一起被装载到一个匿名数组中，然后提供给 `Storeable::store_fd`。

这种实现较慢，不能令人满意(存储 1000 个对象需要一到两秒)。这主要是因为对于每个对象有大量的查询和存取方法调用操作。处于现在这种原型阶段，我不认为这是个问题。

## 获取对象

方法 `load_all`, 由 `new` 调用, 只是完成了与 `flush` 相反的工作。它读取文件, 重新创建每个对象并将其插入属性 `all_instances` 中, 如下所示:

```
sub load_all { # $all_instances = load_all($file);
  my $file = shift;
  return undef unless -e $file;
  open(F, $file) || croak "Unable to load $file: $!";
  # 首先是全局信息
  my ($class, $id, $obj, $rh_attr_names, @attrs, $all_instances);
  eval {
    while (1) {
      ($class, $id, @attrs) = @{Storeable::retrieve_fd(\*F)};
      $obj = $all_instances->{$id};
      $obj = $class->new() unless defined($obj);
      $rh_attr_names = $this->get_attrs_for_class($class);
      $obj->set_attributes(
        "_id" => $id,
        map {$rh_attr_names->[$_] => $attrs[$_]}
          (0 .. $#attrs)
      );
      $all_instances->{$id} = $obj;
    }
  };
  $all_instances;
}
```

`load_all` 调用 `Storeable` 的 `retrieve_fd` 函数, 调用相应类的构造函数 (`new`) 来创建那个类的未经初始化的对象, 并在这个新建对象上调用 `set_attributes`。 `map` 语句创建了一组属性的名字值对。当 `Storeable::retrieve_fd` 没有数据可读时, 它会抛出一个例外 (使用 `die`)。该例外将终止无限循环, 但最终会由 `eval` 捕获。

## 查询处理

`retrieve_where` 方法接收一个类名和查询表达式, 后者为 SQL 语法的一个子集。查询并不保证对于诸如 `LIKE`、`BETWEEN` 和 `IN` 等 SQL 关键字能够工作; 然而它对于数据库适配器来说能够工作, 这是因为该表达式不做任何翻译而直接发送给数据库。

编写能够解析和执行任意查询表达式的查询处理程序，不是一件简单的工作。但是我们知道 Perl 自己就处理表达式计算，因此如果我们能够把一条查询转换成 Perl 表达式的话，我们就可以简单的使用 eval 来为我们完成这些棘手的工作了，这同我们在第五章“Eval”中见到的情况类似。

retrieve\_where 于是就调用 parse\_query 来把表达式转换成可计算的 Perl 布尔表达式，并且动态的产生一段包含这个表达式的代码，来遍历属性 all\_instances 中的所有对象。也就是说，一个如下的调用：

```
retrieve_where ('Employee', 'age < 45 && name != 'John')
```

将会被翻译成下面的这段 Perl 代码，然后交由 eval 执行：

```
my $dummy_key; my $obj;
while (($dummy_key, $obj) = each %$all_instances) {
    next unless ref($obj) eq "Employee";
    my ($age, $name) = $obj->get_attributes(qw(age name));
    push (@retval, $obj) if $age < 45 && $name ne 'John';
}
```

push 语句中的布尔表达式和属性名列表都将由 parse\_query 返回。retrieve\_where 的实现如下所示：

```
sub retrieve_where {
    my ($this, $class, $query) = @_;
    my $all_instances = $$this;
    # 空查询将会产生所有对象的列表
    return $this->retrieve_all() if ($query !~ /\S/);

    my ($boolean_expression, @attrs) = parse_query($query);
    # @attrs 包含了查询中使用的属性名
    # 构造一条如下形式的语句来获取需要的属性:
    #   my ($name, $age) = $obj->get_attributes(qw(name age));
    my $fetch_stmt = "my (". join(",", map{'$'. $_} @attrs) . ") = ".
        "\$obj->get_attributes(qw(@attrs))";
    my (@retval);

    my $eval_str = qq{
        my \$dummy_key; my \$obj;
        while ((\$dummy_key, \$obj) = each \%$all_instances) {
```

```

        next unless ref(\$obj) eq "$class";
        $fetch_stmt;
        push (@retval, \$obj) if ($boolean_expression);
    }
};
print STDERR "EVAL:\n\t$eval_str\n" if $debugging ;
eval ($eval_str);
if ($?) {
    print STDERR "Ill-formed query:\n\t$query\n";
    print STDERR $@ if $debugging;
}
@retval;
}

```

不是为每个查询都创建一个对象列表，`retrieve_where` 应当有选择的以一个回调引用作为第三个参数，它对于每一个匹配该查询的对象都会被调用。

现在让我们来看一下 `parse_query`，像我们前面提到的那样，它将 SQL `where` 子句翻译成一个 Perl 表达式。输入的查询表达式根本上是一系列形为“变量 操作符 值”的查询短语，其间用逻辑表达式（`&&` 和 `||`）连接起来。进行转换的规则如下所示：

1. 如果查询为空，那么它应当计算为 `TRUE`。
2. 转义引号应当保持不变。也就是说，诸如 `"foo\'bar"` 的字符串不应该造成混淆。
3. `"=` 映射到 `"=="`。
4. `variable` 被映射到 `$variable`。在进行这一步处理时，`parse_query` 同时还要记录保存所碰到的属性名。该列表将返回给调用过程 `retrieve_where`。
5. 如果 `value` 是括起来的字符串，那么 `op` 将映射到相应的字符串比较操作符（看下面的 `%string_op`）。

`parse_query` 的实现如下：

```

my %string_op = ( # 将任何操作符映射成相应的字符串形式
    '==' => 'eq',
    '<'  => 'lt',
    '<=' => 'le',

```

```

        '>'    => 'gt',
        '>='   => 'ge',
        '!= '  => 'ne',
    );
my $ANY_OP = '<|=|>|<|>|!=|==';      # 任何比较操作符
sub parse_query {
    my ($query) = @_;
    # 规则 1
    return 1 if ($query =~ /^\\s*$/);
    # 首先将经过转义的引号实例暂存起来 - 规则 2
    # 这样在我们进行处理时就不会碍事了
    # 规则 5.
    $query =~ s/\\[# 希望没有使用 \\200 和 \\201
    $query =~ s/\\[\"]/\\201/g;
    # 规则 3  将所有的 '=' 替换成 '=='
    $query =~ s/([^\>=<])=/$1 == /g;
    my %attrs;
    # 规则 4  抽取字段并将 var 替换成 $var
    $query =~
        s/(\\w+)\\s*($ANY_OP)/$attrs{$1}++, "\\$1 $2"/eg;
    # 规则 5  在括起来的字符串前将比较操作符替换成字符串形式的比较操作符
    $query =~
        s{
            ($ANY_OP)          (?# 任何比较操作符)
            \\s*                (?# 后跟 0 个或多个空白符)
            ['(?# 然后接着是一个括起来的字符串 )
        }{
            $string_op{$1} . '\\'. $2 . '\\\"
        }goxse;  # 全局的, 编译一次的, 扩展的
                  # 当作单一的一行, eval
    # 恢复所有经过转义的引号字符
    $query =~ s/\\200/\\'/g;
    $query =~ s/\\201/\\\"/g;
    ($query, keys %attrs); # 返回修改过的查询及字段列表
}

```

## Adaptor::DBI

Adaptor::DBI 要比 Adaptor::File 简单的多。它并不在内存中维护一张对象表。当要求存储一个对象时，它就会将其发送到数据库中。而当要求获取一个或多个对象时，它也是简单的将请求传递给数据库。这种模式也是它最大的缺点，正如我们在前面一节“对象在内存中的唯一性”中所指出的那样。



如第十章“持续性”中所描绘的那样，方法new简单的打开一个DBI连接并创建一个以该连接句柄为主要属性的适配器对象。这里没有什么高深的理论。

## 存储对象

适配器的store方法把一个对象发送到数据库：

```
sub store {    # adaptor->store($obj)
    (@_ == 2) || croak 'Usage adaptor->store ($obj)';
    my $sql_cmd;
    my ($this, $obj) = @_;
    my $class = ref($obj);
    my $rh_class_info = $map_info{$class};
    my $table = $rh_class_info->{"table"};
    croak "No mapping defined for package $class"
        unless defined($table);
    my $rl_attr_names = $rh_class_info->{"attributes"};
    my ($id)           = $obj->get_attributes('_id');
    my ($attr);
    if (!defined ($id)) {
        $id = $this->_get_next_id($table);
        $obj->set_attributes('_id'=> $id);
        # 生成如下的语句:
        #      insert into Employee (_id, name, ' age)
        #      values (100, "jason", 33)
        $sql_cmd = "insert into $table (";
        my ($col_name, $type, $attr);
        my (@attrs) = $obj->get_attributes(@$rl_attr_names);
        $sql_cmd .= join(", ", @attrs) . ") values (";
        my $val_cmd = "";
        foreach $attr (@attrs) {
            my $quote = ($attr =~ /\D/)
                ? "\""
                : "";
            $val_cmd .= "${quote}${attr}${quote}, ";
        }
        chop ($val_cmd);
        $sql_cmd .= $val_cmd . ")";
    } else {
        # 对象在数据库中已经存在。更新它
        # 使用如下的语句:
        #      update Employee set name = "jason", age = 33
        #      where _id = 100;
```

```

$sql_cmd = "update $table set ";
my ($name, $quote);
my @attrs = $obj->get_attributes(@$rl_attr_names);
foreach $name (@$rl_attr_names) {
    if ($name eq '_id') {
        shift @attrs; # Can't update primary row
        next;
    }
    $attr = shift @attrs;
    $quote = ($attr =~ /\D/)
        ? ""
        : "";
    $sql_cmd .= "$name=${quote}${attr}${quote}, ";
}
chop($sql_cmd); # remove trailing comma
$sql_cmd .= "where _id = $id";
}
# 构造好 SQL 查询语句，将其提交给相应的数据库连接执行
$this->{dbconn}->do($sql_cmd); #
die "DBI Error: $DBI::errstr" if $DBI::err;
$id;
}

```

全局变量 %map\_info 为配置文件中提到的每个包保存了数据库配置信息，其中包括：相应数据库的表名，持续性属性列表和对应的数据库列名。如果对象已经有了一个名为 \_id 的属性，相应的数据库行将会被更新。否则，将分配一个新的标识并插入一条新的数据库记录。所有拥有字符串值的属性将被自动的括起来。

显然，我们可以比这个实现做的更好。如果我们创建 1000 个对象，那么前面的代码将会创建和计算 1000 条全新的 SQL 插入语句。一种更好的解决方案是在第一次碰到该类对象时，预先准备好 insert/delete/update/fetch 语句，如下所示：

```

$insert{'Employee'} = $dbh->prepare (
    "insert into Employee (_id, name, age)
    values ( ? , ? , ? )");
$delete{'Employee'} = $dbh->prepare (
    "delete from Employee where _id = ?";
$update{'Employee'} = $dbh->prepare (
    "update Employee (name=?, age=?");
$fetch {'Employee'} = $dbh->prepare (

```

```

        "select name, age, from Employee
           where _id = ?");

```

store 只需根据相应的语句来执行这些语句。一种更快的方式是利用存储过程。即便这样，对于原型系统来说当前的实现就工作的挺好。

附带说一下，Adaptor::DBI 的 flush() 方法什么也不做，因为 store() 并没有把对象保存在内存中。

## 查询

retrieve\_where 根据类的映射信息创建一个 select 查询。正如前面所指出的那样，同样的查询执行两遍将会得到两组不同的对象，这是两组重复的数据：

```

sub retrieve_where {
    my ($this, $class, $query) = @_;
    my $where;
    $where = ($query =~ /\S/)
               ? "where $query"
               : "";
    my $rh_class_info = $map_info{$class};
    my $table = $rh_class_info->{"table"};
    croak "No mapping defined for package $class"
        unless defined($table);
    my $rl_attr_names = $rh_class_info->{"attributes"};
    my $rl_col_names = $rh_class_info->{"columns"};
    my $sql_cmd = "select"
                  . join(", ", @{$rl_col_names})
                  . "from $table $where";
    print $sql_cmd if $debugging;
    my $rl_rows = $this->{d}->do($sql_cmd);
    my @retval;
    my $size = @{$rl_attr_names} - 1;

    if ($rl_rows && @{$rl_rows}) {
        my $i; my $rl_row;
        foreach $rl_row (@$rl_rows) {
            my $obj = $class->new;
            $obj->set_attributes(map {
                $rl_attr_names->[$_] => $rl_row->[$_]
            }(0 .. $size));

```

```
        push (@retval, $obj);
    }
}
@retval;
}
```

前面的 `set_attributes` 语句或许需要解释一下。这条语句的目的是设置由数据库返回的所有属性。由于 `set_attributes` 要求名字-值对的列表，因此我们使用内建的 `map` 函数来返回一个列表。这个函数需要两个参数——一个代码块和一个列表，并且对列表中的每个元素，在列表的上下文环境中计算这个代码块。该函数返回包含所有迭代所执行代码块的结果的列表。

此时此刻，如果你的兴致未减，你会发现，非常值得回过头去理解一下适配器如何处理在“设计注意事项”一节中所提出的问题。

## 相关资源

下面的图书和万维网站有许多有关对象持续性的很好的论述：

1. Object-Oriented Modeling and Design. James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, 和 William Lorensen. Prentice-Hall, 1991。

对在关系数据库中实现面向对象模型的优秀论述。

2. Object Persistence. Roger Sessions. Prentice Hall, 1996。

有关 CORBA 持续性体系结构的讨论。讨论这些委员会背后的政治因素与讨论技术资料一样有趣。

3. CORBA Persistence Service Specification, Object Management Group (OMG)。  
位于 <http://www.omg.org>。

4. PJava: Orthogonal Persistence for Java。

位于 <http://www.dcs.gla.ac.uk/pjava/>。特别的要查看一下题为“Design Issues for Persistent Java: a Type-Safe, Object-Oriented, Orthogonally Persistent System”的设计材料。

5. Object Database Management Group (ODMG): <http://www.odmg.org>。
6. Taming C++: Pattern Classes and Persistence for Large Projects. Jiri Soukup. Addison-Wesley, 1994。
7. Equal Rights for Functional Objects, or, The More Things Change, The More They Are the Same. Henry Baker。

这份资料将告诉你, 讲述对象标识的内容比眼前的要多得多。可以从以下地址下载: <ftp://ftp.netcom.com/pub/hb/hbaker/ObjectIdentity.html>。



## 第十二章

# 使用套接字进行网络编程

### 本章简介:

- 网络计算入门
- Socket API 和 IO::Socket
- 同时处理多个客户端
- 现实世界中的服务器
- IO 对象与文件句柄
- 预编译的客户端模块
- 相关资源

我把电话插在过去搅拌机所在的地方。我打电话给某个人。他们于是就发出“啊啊啊……”的声音。

——Steven Wright

程序之间可以通过各种方式进行通信。它们可以使用文件，匿名/有名管道，System V 进程间消息原语，BSD Socket（套接字）和 TLI（传输层接口）等。套接字和 TLI 正符合网络计算的含义，要比其他的 IPC（进程间通信）机制的意义更近一层，因为它们并不限制相互通信的进程要在同一台机器上。本章将会提供有关套接字通信的入门知识，并且还将使用 Graham Barr 的 IO 库（标准 Perl 发行版的一部分）来创建简单的客户/服务器架构。我们将在下一章应用本章所讲的知识，创建一个异步消息传输模块和进行远程过程调用（RPC）的模块。

网络计算是我们在本书中所讨论的四种重要技术的第二种；另外几种为用户界面、持续性存储和代码生成。我们在这一章，将像讨论其他三种技术时一样，在关注 Perl 对此所提供的支持同时也同样关注技术本身。Andrew Tanenbaum 的有关计算机网络的教科书《Computer Networks》是一部精彩的计算机网络入门教材（我把它评价为有史以来写的最好的计算机图书之一）。本章只提供够用的与 Perl、套接字和 TCP/IP 相关的网络入门知识。

## 网络计算入门

邮件（纸质的或电子的）和电话是两种不同形式的通信方式。电话交谈是面向连

接的，因为呼叫者与被叫者之间将“占有”一条线路（一个连续的连接），直至通话结束。面向连接的通信可以保证消息可靠的发送，保持消息发送的顺序，而且可以以数据流的形式进行传输。相比之下，邮件是一种无连接模式的传输，它以包（或数据报）的形式传输信息，而且不能保证消息的可靠发送和每个包接收的顺序。由于每个包都标识它的发送者和接收者，因此有更高的开销；而对于面向连接的会话来说，一旦双方互相确认以后，不用罗嗦这么多就可以进行传输。计算机网络同样为你提供了相似的选择：有连接的和无连接的数据传输。我们必须提到的一点是，存在诸如可靠的UDP的无连接协议而且确实可以保证发送和序列的完整性。

在网络计算世界中，每一台计算机都被分配一个因特网地址，也被称作 IP（Internet Protocol 的缩写）地址。这是一个四字节序列，通常以点分的形式来书写，如 192.23.34.1。（这种格式将会随着 IP v6 的到来而改变，因为我们这个世界几乎快速耗尽了四字节的 IP 地址空间。）正如你的电话有方便的别名一样，如 1-800-FLOWERS，计算机也经常被授予唯一的别名，如 www.yahoo.com。现在在一个机器上可以运行多个程序，只将消息发送到机器是不够的，它必须转交给运行在机器上的相应的应用程序。一个程序可以要求打开一个或多个端口，这等同于一个私有邮箱或电话分机。要将消息发送到某个程序，你必须有它完整的地址：它的机器名和它所监听的端口号。那些标准的诸如 ftp, telnet 和 mail 的应用实际上以成对的形式出现；比如，你使用 ftp 程序与位于远端机器上对应的服务器程序 ftpd（ftp 守护进程）进行通话。这样的服务器程序在标准端口上监听；当你在浏览器上键入 www.yahoo.com 时，浏览器将会自动的与那台机器的端口 80 进行连接，它假定相应的 Web 服务器就在那里监听。端口号 1-1024 保留由标准的，众所周知的因特网应用来使用。许多平台都保留使用名字“localhost”（和地址 127.0.0.1）来表示程序正在运行的机器。

每当分配一个套接字时，你的程序都可以选择使用称作 TCP/IP（传输控制协议/IP）的面向连接的协议，或者使用无连接的 UDP/IP（用户数据报协议）协议。显然发送方与接受方必须使用相同的协议。TCP/IP 模型通常更倾向于使用 TCP 协议，因为它提供数据的有序传输，端到端的可靠性（校验和、肯定的确认和超时）和端到端的流量控制（如果发送方发送数据的速度大于接收方的处理能力，它将在接收方的缓冲区满时使发送方阻塞）。如果通信媒介非常好，比如局域网，那么

使用 UDP 将会获得更好的性能，因为它不必考虑最坏的情况。然而在生产系统中，你决不能凭运气做事，因此我们将在本章中坚持使用 TCP。

套接字抽象和 API 首先在 BSD 4.2 中引入，以为不同类型的传输协议（除了 TCP 和 UDP 外还有别的协议）提供统一的界面，而且根据所使用的协议，一个套接字即可以像电话接收器也可以像邮箱那样工作。对于任何一种情况，要建立会话都需要在每一端有一个套接字。（这也是为什么套接字也被认为是通信端的原因。）socket API 允许你指定通信实体的域——“Unix 域”适用于同一台机器中的进程，而“Internet 域”则适用于不同机器上的进程。这一章我们将查看更为通用的（和有用的）“Internet 域”选项。

TLI 是另一种在 System V (Release 3.0) 中引入的 API，它提供了与套接字抽象类似的另一种方案，但是由于它不像 BSD 套接字接口这样应用广泛，我们在这一章中不予讨论。

## Socket API 和 IO::Socket

Perl 提供对套接字的本地支持，并且还提供了一个名为 Socket 的模块来简化与本地套接字调用有关的工作。然而仍然有许多选项需要处理。由于大部分的应用只使用一小部分标准的选项，我们将取而代之使用一种真正方便的称作 IO::Socket 的模块，该模块是建立在套接字之上的。

在这一节，我们将使用这个模块来建立一个发送和接收程序。

### 接收者

就像你会向电话公司索要电话号码和电话机一样，发送方和接受方也都会要求模块为它们创建套接字。套接字像电话一样是双向通信端：一旦连接建立起来后，每一方都可以发送和接收数据，只需双方在两个程序之间的通信方向上相互理解。

因为只需接收方需要有一个众所周知的地址，我们可以像下面这样创建接收套接字：



```
use IO::Socket;
$sock = new IO::Socket::INET (LocalHost => 'goldengate',
                              LocalPort => 1200,
                              Proto      => 'tcp',
                              Listen     => 5,
                              Reuse      => 1,
                              );
die "Could not connect: $!" unless $sock;
```

IO::Socket::INET模块为因特网域的套接字提供了一个友好的包裹层。LocalHost和LocalPort参数分别用来指定主机和套接字要监听的端口号。这里任意挑选使用1200，但是你必须确保它不与位于这台机器上的其他应用所使用的端口号冲突。（否则，你会得到错误信息，“地址已被使用”。）我们设置了Reuse选项，这是因为如果该程序在没有恰当的关闭套接字之前，就退出后又重新启动执行，它会抱怨套接字已经使用的错误信息。打个比方说，Listen选项指定在拨打这个号码时可以挂起等待的最大呼叫者数。

一旦套接字建立起来以后就准备好了接收打入的呼叫了。accept()方法一直在给定的端口上监听，直到有另一个程序试图连接到它为止（我们马上就要看到拨打方是如何完成这项工作的）；这时accept会返回一个新的套接字：

```
$new_sock = $sock->accept();
```

这与电话交换机接线员让你通过另一部话机继续交谈一样，而他则继续接听从主要号码打入的电话。由客户端发送的消息现在可以通过\$new\_sock来获得。你可以把这个套接字用作文件句柄，并能够调用任何输入操作符，<>，read，或sysread，如下所示：

```
$buf = <$new_sock>;
# 或者
$bytes_read = sysread ($new_sock, $buf, $num_bytes_to_read);
```

在文件读到文件末尾时它们均会返回undef。

下面的代码总结了一下我们上面所讨论的内容。它将一个套接字绑定到一个地址上并等待接入连接请求。当有请求发生时，它会从新建的套接字中读取数据直到

另一端关闭连接为止。此时操作符 <> 则会返回 undef (sysread 返回 0, 读取的字节数)。

```
use IO::Socket;
$sock = new IO::Socket::INET (LocalHost => 'goldengate',
                              LocalPort => 1200,
                              Proto      => 'tcp',
                              Listen     => 5,
                              Reuse      => 1
                              );
die "Socket could not be created. Reason: $!" unless $sock;
while ($new_sock = $sock->accept()) {
    while (defined ($buf = <$new_sock>)) {
        print $buf;
    }
}
close ($sock);
```

你也可以使用 \$new\_sock->get\_line() 来替代 <\$new\_sock>。

## 发送者

呼叫方的代码更加简单。它创建一个套接字并指定接收端的地址, 如果成功的话, 它就会开始向它发送数据:

```
use IO::Socket;
$sock = new IO::Socket::INET (PeerAddr => 'goldengate',
                              PeerPort => 1200,
                              Proto     => 'tcp'
                              );
die "Socket could not be created. Reason: $!\n" unless $sock;
foreach (1 .. 10) {
    print $sock "Msg $_: How are you?\n";
    $sock->flush();
}
close ($sock);
```

注意对于服务器端或客户端来说, 提供给 IO::Socket::INET::new() 方法的参数有什么不同。Listen 和 Reuse 参数对于发送端来说将被忽略。

## 双向通信

你可以从套接字中读或者写，但是前面的脚本程序表明，两个通信的进程必须达成一种共识，到底哪一方在讲和哪一方在听。打个比方来讲，如果双方都太礼貌并开始读取各自的套接字(`sysread`和其他的输入操作符将会等待直到能够读取要求数量的数据为止)，那么程序就会死锁。同样，如果双方都不礼貌而同时开始对着电话讲话(`syswrite`在缓冲区满时会挂起，这是因为另一方没有在看，死锁是完全可能的)，同样会导致死锁。在一个典型的客户/服务器环境中，这种协议是制定好了的。客户端程序开始建立会话，它发送一个请求并等待回应。典型的服务器永远不会企图与客户端程序进行连接或建立会话；它只会监听并作出回答。死锁只会发生在端与端对等的会话中。

## 同时处理多个客户端

`accept`、`read`和`sysread`是阻塞式调用这一事实，对服务器来说有着更多的影响(注1)。一个单一线程的进程每次只能调用一个这样的调用，如果没有太多的客户端来请求服务器的响应，并且没有客户端过多的占用服务器的时间的话，就不会有什么问题。现实世界是可怕的，你必须解决这些问题。有三种方式来解决这个问题：

1. 创建多道控制(多个进程或多个线程)，并让每个调用在自己的那道执行中阻塞。
2. 只在你确信它们不会阻塞时才执行调用。我们称这种方式为“select”解决方案，因为我们通过使用 `select` 调用来保证某个套接字有东西要提供。
3. 使用 `fnctl` 或 `ioctl` 来使这些套接字变成非阻塞的。

我们马上就要看到，在生产系统中，第二种选择方案应当与第三种结合起来使用。在所有情况下，在我们尝试这些选择时，客户端代码将不受影响。

附带说一下，还有第四种解决方案。一些系统支持一种异步 I/O 通知机制：如果

---

注1: `accept` 阻塞到有人试图进行连接时为止。

有套接字准备好进行 I/O 操作，信号 SIGIO 将会发送给这个进程。我们不来谈论这种方案，因为对于信号处理程序来说，它不知道究竟是哪一个套接字准备好读或写了。

## 多道执行

Perl 现在还不支持线程（至少还没有正式支持它）（注 2），但是在 Unix 和具有相当能力的系统中，都支持 fork，一种获得进程级并发性能的方式。服务器进程充当一种全职的接待员：它在 accept 上阻塞，并且在一个连接请求到来时，它产生一个子进程然后继续执行 accept。与此同时，新建的子进程拥有其父进程环境的拷贝并共享所有打开的文件句柄。于是它能够对 accept 返回的套接字进行读或写。在子进程完成了会话任务之后，它就会简单的退出。因此每个进程都专注于它自己的任务，而且相互之间互不干扰。下面的代码描述了一个 forking 服务器的例子：

```
#Forking 服务器
use IO::Socket;
$SIG{CHLD} = sub {wait ()};
$main_sock = new IO::Socket::INET (LocalHost => 'goldengate',
                                   LocalPort => 1200,
                                   Listen    => 5,
                                   Proto     => 'tcp',
                                   Reuse     => 1,
                                   );
die "Socket could not be created. Reason: $!\n" unless ($main_sock);
while ($new_sock = $main_sock->accept()) {
    $pid = fork();
    die "Cannot fork: $!" unless defined($pid);
    if ($pid == 0) {
        # 子进程
        while (defined ($buf = <$new_sock>)) {
            # 操作 $buf ....
            print $new_sock "You said: $buf\n";
        }
        exit(0); # 子进程在完成时退出
    }
}
```

注 2：Malcolm Beattie 有一个线程版 Perl 解释器的工作原型，它将会在 Perl 5.005 发行版被集成到主流代码中。

```
    ) # 否则它是父进程，它将返回继续执行 accept()
}
close ($main_sock);
```

**7** `fork`调用从跟在`fork`后面的语句开始，会产生两个相同的进程——父进程和子进程。父进程得到一个正的返回值，也就是子进程的进程ID号（`$pid`）。两个进程都将检查这个返回值并执行各自的逻辑控制；主进程将返回继续执行`accept`，而子进程则从套接字中读取一行信息并将其回送给客户端。

附带说一下，信号`CHLD`与`IPC`本身没有任何关系。在Unix系统中，当一个子进程退出时（或非正常终止时），系统会清楚占用的内存，文件和其他与之关联的资源。但是还要保留一小部分信息（包括退出状态，如果子进程能够执行`exit()`的话；对于其他别的情况是结束状态），这仅仅是为了万一父进程使用`wait`或`waitpid`来查询它的状态时用。这个终止的子进程也被称做僵尸（*zombie*）进程，使用`wait`来将其清除掉总是一个好习惯；否则，进程表将会充满垃圾。在前面的代码中，`wait`并不阻塞，因为`CHLD`信号已经为我们安排好了，只有在有进程退出时才调用它。一定要阅读联机文档来澄清与信号相关的quirk的整体概念，特别是`SIGCHLD`等有关的问题。

## 使用 select 来完成多路传输

我们在前面一节当中创建另外不同进程的原因，就是为了避免在调用`accept`、`read`或`write`时导致阻塞，而忽略在其他套接字上进行的工作。我们还可以使用从BSD Unix中引入的`select`调用，它将在某个套接字（实际上是任何句柄）可以读出或写入时返回控制。这种方法可以允许我们使用一种单一执行进程的方案——这多少有点像把接待员给解雇了，然后由我们自己来处理所有的拨入呼叫和会话。

本地`select`调用所提供的接口不怎么美观，因此我们使用`IO::Select`包裹模块来代替。如下所示：

```
use IO::Socket;
use IO::Select;
$sock1 = new IO::Socket (...);
$sock2 = new IO::Socket (...);
```

```

$read_set = new IO::Select;
$read_set->add($sock1);
$write_set = new IO::Select;
$write_set->add($sock1, $sock2);

```

IO::Select 模块的 new 方法创建一个代表文件句柄集合的对象，并使用 add 和 remove 来对其进行修改。select 方法（它将调用 Perl 的本地 select 函数）接受三种文件句柄集合或 IO::Select 对象，并对它们的可读性、可写性和错误状态分别进行监控。在前面的代码段中，我们创建了两个集合——如果你愿意的话，文件句柄可以被加入到其中任何一个集合中——然后像下面这样把它们提供给 select 方法：

```

($r_ready, $w_ready, $error) =
    IO::Select->select($read_set, $write_set, $error_set, $timeout);

```

Select 将被阻塞直到有意的事件（一个或多个文件句柄已经准备好了可以进行读、写或报告错误状况）发生或超时值已到时为止。这时，它会创建三个分离的已经准备好的文件句柄的列表，并返回指向它们的引用。超时值以秒为单位但是你可以使用浮点数来表达以获得毫秒级的精度。

让我们使用这些信息来实现一个从一个或多个客户端获取信息的程序：

```

# 像以前一样创建主套接字 ($main_socket)...
# ....

use IO::Select;
$readable_handles = new IO::Select();
$readable_handles->add($main_socket);
while (1) { #Infinite loop
    # select() 将阻塞到有一个套接字可读或可写为止
    ($new_readable) = IO::Select->select($readable_handles,
                                         undef, undef, undef);
    # 如果运行到这里，表明至少存在一个可以读取的句柄
    # 此刻我们只操心读的方面
    foreach $sock (@$new_readable) {
        if ($sock == $main_socket) {
            $new_sock = $sock->accept();
            # 将它加入到列表中并返回执行 select
            # 因为这个新建的套接字可能还没有什么可读取的

```

```
        $readable_handles->add($new_sock);
    } else {
        # 它是一个普通的客户端套接字，准备好了进行读取
        $buf = <$sock>;
        if ($buf) {
            # .... 操作 $buf
        } else {
            # 客户端关闭套接字。我们在此也这么做，并将其
            # 从 readable_handles 列表中删除
            $readable_handles->remove($sock);
            close($sock);
        }
    }
}
}
```

我们创建了一个监听套接字，\$main\_socket并将其配置为在一个知名端口上进行监听。我们然后把这个套接字添加到一个新建的IO::Select集合对象中。当select第一次返回时，\$main\_socket有东西需要读出（或者有一个错误，我们暂且忽略这种可能性）；换句话说就是它收到了一个连接请求并保证在调用accept时不会被阻塞。现在，如果从accept返回的套接字没有什么东西可读我们可不愿因它而阻塞，因此我们把它加入到将被进行可读性监控的文件句柄列表中。在select下一次返回时，我们知道这两个套接字中的一个已经准备好读了（或者两个都准备好了）。如果准备好的是 \$main\_socket，那么我们将重复上面的动作。否则，我们就有了一个可以读取信息的套接字。

当一个或多个远程的套接字关闭时，select同样要返回。在执行了I/O操作之后，位于监听端的相应的套接字将返回0（读或写了0字节）上面的服务器将这些套接字从IO::Select集合中删除以防止select每次都返回同样的已经死亡的套接字。

## 阻塞又发生了

我们本节所要做的就是依赖select来告诉我们一个文件句柄已经准备好了读或写操作，然后才真正的进行读写。不幸的是，我们不知道到底在I/O缓冲区中已经积攒了多少数据（为了进行读操作），或者可以向它写入多少数据（另一端或许读的很慢，于是你这一端要输入的数据量是多少）。sysread和syswrite均返回实际读或写的字节数，因此你必须在循环中调用它们，直至所有的信息都

读写完毕。你一旦读空了（或者填满了）缓冲区，确实存在这种可能性，就是它仍有可能在对方稍慢一拍的情况下阻塞下一次读或写的企图。一种解决方案就是，在每次循环中调用 `select`，并且只在它确认套接字可用时才进行下一步操作。当文件句柄同时要应付你的读和写请求时，这种方法会减慢程序的执行速度。况且你还要在 `select` 告知你文件句柄不可用，并在以后文件描述符改变状态时，退出循环。

对于单一线程的程序来说，另一种选择就是使文件句柄变成非阻塞的。请接着往下读。

## 非阻塞式文件句柄

通过使用特定于操作系统的 `fcntl` 或 `ioctl` 调用，任何文件句柄都可以变成非阻塞的，如下所示：

```
use POSIX;
fcntl($sock, F_SETFL(), O_NONBLOCK());
```

`Fcntl` 模块（文件控制）使文件 `fcntl.h` 中的常量可以作为函数来使用。`fcntl` 函数接收一个诸如 `F_SETFL`（“set flag”）的命令和一个特定于该命令的参数。根据操作系统的不同，用于设置非阻塞式 I/O 的标志还可能为 `O_NDELAY` 或 `FNDELAY`。

任何情况下，该操作一旦执行，如果 `sysread` 和 `syswrite` 不能马上完成相应的操作，都将立即返回 `undef`（不是 0）并将 `$!` 设置为 `EAGAIN`（或者在 BSD 4.3 上为 `EWOULDBLOCK`）。下面的代码描述了读套接字时，这些返回和错误值的情况：

```
# 希望读取 1024 个字节
$bytes_to_read = 1024; $msg = '';
while ($bytes_to_read) {
    $bytes_read = sysread($sock, $buf, $bytes_to_read);
    if (defined($bytes_read)) {
        if ($bytes_read == 0) {
            # 远端套接字关闭了连接
            close($sock);
            last;
        } else {
            $msg .= $buf;
            $bytes_to_read -= $bytes_read;
        }
    }
}
```



```
    }  
  } else {  
    if ($! == EAGAIN()) {  
      # 可以返回执行 select。这里我们选择  
      # 轮循等待可以进行读取事件的发生  
    } else {  
      last;  
    }  
  }  
}  
}
```

一种简单的可选方案是不使用 `select` 调用，而在一个简单的轮循中对每一个套接字调用 `read` (或 `sysread`) (或在主套接字上调用 `accept`) 来检查它是否有数据要读，或是如果我们要写数据的话，调用 `write` (或 `syswrite`)。这样就不必担心阻塞的发生。这种方案对 CPU 资源来讲是一种持续性的消耗，因为该进程根本不会处于空闲 (*quiescent*，注 3) 状态。用客户/服务器的说法就是你应当总是设法去建造一个安静的服务器。

你可能已经注意到了，我们在所有这些讨论中都忽略了客户端的情况。如果客户端愿意被阻塞，那么就不成问题，由于不同于服务器端，它不会与多于一个的实体交谈。但是如果它包含有 GUI 的话，显然将无法承受阻塞的发生，这样我们就遇到了同样的问题。我们将在第十四章“使用 Tk 进行用户界面编程”中重新来探讨这个问题。在没有对“客户”与“服务器”有清晰界定的系统中——如银行计算机的集群系统就是这样一个对等系统的例子——每个进程都建立在我们前面所讨论的通用的服务器架构之上。

你现在可以看出所有这三种创建服务器的方案都有其蹩脚和缺憾之处。在下面的章节中我们将介绍应用在典型生产系统中的技术和策略。

## 现实世界中的服务器

单一线程的服务器本质上是事件驱动的——它们通过响应超时和 I/O 事件来执行程序。它们一般不会为每个请求花费太多的 CPU 时间，这是因为它们需要返回

---

注 3：《Webster's Tenth Collegiate Dictionary》将此词解释为“具有明显的不活动或静止的特征”。

到 `select` 中来处理其他的事件，这些事件可能此时已经排起队了。大多数生产用的单一线程服务器还使用非阻塞文件句柄（将在“同时处理多个客户端”一节中列出的第二和第三个选项进行组合）。在下一章中我们将使用这些技术来创建一个小型的消息传递功能库。使用单一线程的好处在于，处理那些频繁而周期短小的请求开销非常小。而且数据结构可以很容易的在所有并发的会话间共享或者进行高速缓存，以供将来的会话使用。例如聊天服务器就非常适合这种体系结构。

当服务器无法保证一个给定的请求需要花费多长时间来处理时，我们会选择多进程方案。Web 服务器就是使用的这种方案，它简单的产生一个 CGI（公共网关界面）程序来处理与另一端 Web 浏览器的会话。现今的趋势是将快速的任务交由 Web 服务器自身来处理，而只有当该项任务有可能将整个服务器挂起时，才产生子进程。当然问题就出在产生子进程开销较大，于是一种流行的选项就是预先生成一定数量的进程，并在请求到来时将任务再转交给它们。显然如果套接字的数量远大于预先生成的进程数量，父进程除了使用 `select` 来复用它们外别无选择。如你所见，前面一节描述的选项决不是相互独立的。

如果环境支持的话（Perl 还不支持），多线程也是种选择。Java 热衷于这种实现方式并且预期在线程上的 I/O 阻塞；实际上，它甚至没有提供一个 `select` 的接口。这种方案的好处就是它要比多进程的模式更为轻快。同时你还获得了并发性能和数据共享。不利之处就是典型的工作站一般在你引入 40 或更多的内核级线程时，性能会急剧下降，因此它们只能支持有限数量的客户端。Solaris 上的线程要好的多，因为它们区分轻量级、用户级和内核级线程。无论如何，这种选择对于当前的 Perl 程序员来说无法使用，因此讨论也没有什么意义。

## IO 对象和文件句柄

Perl 支持 BSD 的套接字调用，它将返回一个文件句柄，这与 `open` 对文件和管道所完成的任务相同。该文件句柄可以被用作所有内建输入输出操作符：`<>`、`read`、`sysread`、`print`、`write`、`syswrite` 等等的参数。同时它还能被与套接字相关的函数如 `send`、`recv` 和 `setsockopt` 使用。

IO::Socket模块的new方法所返回的对象，同样可以被用做这些I/O例程的参数。在内部，它调用套接字并使用与该文件句柄对应的typeglob来存储其他的属性。我们已经在第八章“面向对象：下面的几步”中的“高效的属性存储”一节里，描述了这种丑陋的技巧。换句话说，它的返回值就是传递给套接字的对象。这也是为什么你选择哪个选项对I/O操作符来说都无关紧要的原因。我建议大家使用相当简单的IO::Socket选项。

然而IO::Select则是另一回事。如果性能非常关键，你也许更愿意自己来完成IO::Select所实现的工作：

```
$r_bitset = $w_bitset = $e_bitset = '';  
# 监控 $sock1 的可读性  
vec($r_bitset, $sock1->fileno(), 1) = 1;  
# 监控 $sock2 的可读性  
vec($w_bitset, $sock2->fileno(), 1) = 1;  
# 监控二者的错误信息  
$e_bitset = $r_bitset | $w_bitset;  
  
($nfound, $timeleft) =  
    select ($r_bitset, $w_bitset, $e_bitset, $timeout);
```

本地的select函数要求用三位的向量来表示打开的文件、套接字或管道的集合。这些位集中的每一位对应于一个整型的文件描述符，而它则反过来由相应的文件句柄或IO对象来追踪。IO::Socket的fileno()方法或内建函数fileno可以被用来存取这个数字。剩下的工作就简单了：我们创建三个位集来检查可读性，可写性和错误状态，并使用vec来设置位集中的相应位。在select返回之前，它将会修改位集来表示哪个文件描述符已经准备好了进行输入或输出操作。

由于这些位集被修改了，因此我们必须在返回select之前重新构建它们，这样做有点开销过大。相反一个常用的技巧就是在让select操作这些位集之前先做一个拷贝：

```
# Set up $r_bitset and $w_bitset once  
...  
while (1) {  
    ($nfound, $timeout) = select ($r_copy = $r_bitset,  
                                  $w_copy = $w_bitset,
```

```
                                $e_copy = $e_bitset, $timeout);  
    # 检查 $r_copy, $w_copy 的可读性 ...  
}
```

注意赋值发生在select获得控制以前,而select看到的只是\$r\_copy、\$w\_copy和\$e\_copy,这些它可以随意修改。

唯一一个比使用 IO::Select 真正节省时间的地方,就是我们不必创建准备好的文件句柄列表;我们可以直接处理位集。对于我所建立的应用来说,这种微弱的效率提升划不来,因此我将会使用 IO::Select。

## 预编译的客户端模块

诸如邮件客户端、FTP、Web浏览器、telnet和新闻组阅读器等应用程序,都要使用TCP/IP和套接字。位于CPAN上的几个功能库为你提供了客户端的库函数,可以使你在编制出自己的FTP或邮件前端时,无须担心应用层协议。(注意不存在允许你编写自己的服务器来处理这些协议的库。)在这一节我们要来简要的了解几个有趣的客户端模块,它们打包位于Net下面,并且可以以libnet的形式从CPAN获得。这些包也是由Graham Barr编写的。

### Net::FTP

该模块实现了客户端的文件传输协议,用法如下:

```
use Net::FTP;  
$ftp = Net::FTP->new("ftp.digital.com");  
die "Could not connect: $!" unless $ftp;  
$ftp->login('anonymous', 'me@foo.com');  
$ftp->cwd('/pub/plan/perl/CPAN');          # cwd:改变工作目录  
$ftp->get('index');  
$ftp->quit();
```

这个模块支持所有你可以从标准的FTP程序键入的命令。

对于当前的实现而言，`get` 调用将阻塞到整个文件传输完毕，因此尽管对于批处理应用非常有用（如夜间镜像一个 FTP 站点），但是你不能用它来编写图形化的 FTP 客户端。

## Net::POP3

这个库为编程存取 POP（Post Office Protocol，邮局协议）（例如应用于拨号连接）服务器提供了接口。POP 服务器保存发来的 Email，直到邮件阅读器来“访问邮局”为止。让我们来学习一个基于 Net::POP3 的小例子。

大多数基于 PC 的邮件阅读器的毛病在于，它们不让你对消息进行预览，而且不让你决定是否真的要下载其中的邮件。人们总是随意的利用因特网的带宽，你会发现自己等着一封包含麦当娜宝贝近照的邮件缓慢的一点一点流过你的拨号连接而无能为力。这个基于 Perl 的 POP 客户端为位于 POP 服务器上的消息提供了预览功能：它简单的列出所有可用消息的头三行信息：

```
use Net::POP3;
$m = Net::POP3->new('pop.myhost.com'); # POP 服务器名
die "Could not open account" unless $m;
$n = $m->login('sriram', 'foofoo');    # 帐号, 密码
print "Number of msgs received: $n\n";
$r_msgs = $m->list();                  # 返回一个指向散列表的引用, 它将
                                       # msg_id 映射到 msg_size

foreach $msg_id (keys %$r_msgs) {
    print "Msg $msg_id (", $r_msgs->{$msg_id}, "):\n";
    print "-----\n";
    $r1_msg = $m->top($msg_id, 3);      # 获取消息最上面的三行内容
    $, = "\n";
    print @$r1_msg;
}
$m->quit();
```

我使用该脚本程序一个稍微健壮一些的版本，来有选择的删除一些消息，接着启动日常用的邮件阅读器来下载剩余的部分。

## 相关资源

1. perlipc 文档。

覆盖了所有有关 IPC 机制的内容，其中包括套接字通信。

2. Computer Networks, 3rd edition. Andrew S. Tanenbaum. Prentice-Hall, 1996。

最好的一本关于计算机网络的图书。

3. Unix Network Programming. W. Richard Stevens. Prentice-Hall, 1990。

4. Advanced Programming in the Unix Environment. W. Richard Stevens. Prentice-Hall, 1992。

资源  
分享  
PDG

本章简介:

- Msg: 消息传递工具包
- 远程过程调用 (RPC)
- 相关资源

# 第十三章

## 网络计算: RPC 的实现

我等啊等等等，然而却没有任何消息，这时  
我才知道，它一定是从你那里来的。

——Ashleigh Brilliant

这一章，我们将在前面一章所学知识的基础上实现套接字上的两层应用。第一层是一个异步消息传递系统、Msg，它将尽可能的利用非阻塞式 I/O。然后我们在 Msg 层的上面建立一个远程过程调用模块 RPC。RPC 提供方便的同步过程调用，并兼顾了例外处理、wantarray、参数编组等内容。

在我们开始以前先来理解一个基本定义。在第十二章中，我们模糊了“消息”的定义。一个套接字连接只是一串字节流，而由应用程序来定义消息的边界，因此接收端可以知道什么时候一个消息结束了而另一个又开始了。一些协议插入一个消息结束符，这是一个任选的字节如 ASCII 4 (Ctrl-D) 或是包含一个点的孤立的行，而一些则在消息前面缀上消息的长度信息，这样接收端就知道要等待多少数据了。本章将使用后一种选择。

### Msg: 消息传递工具包

本节中，我们使用 IO::Select 和 IO::Socket 模块来实现一个名为 Msg 的事件驱动的，客户服务器类型的消息传递框架（注 1）。下面是它们的关键特性：

注 1： 这里我使用了谜一样玄妙的纯技术语言！

### 排队消息

你可以指示 `Msg` 是立即发送消息还是排队以后发送。

### 非阻塞式 I/O

`Msg` 将检查你的系统是否支持 POSIX，如果支持，它就会使用其对非阻塞 I/O 所提供的支持（如第十二章所讲的内容）。在提供非阻塞 I/O 支持但不兼容 POSIX 的系统中，你可以继承 `Msg` 并重载设置文件句柄阻塞属性的两个方法。而对于根本就不支持这种机制的系统中，`send` 或 `receive` 将会阻塞，但是由于我们使用了 `select` 来判定什么时候是发送或接收消息的最佳时机，它可以把这种调用阻塞的机会降到最低（或长时间阻塞）。

### 消息边界

`Msg` 简单的为每一个发送缓冲区添加4个字节的包含消息长度的前缀。接收端将会预先知道要接收这4个字节，并继而知道消息的长度。

### 消息的透明性

`Msg` 并不费心去查看消息的内容；这就是说在把二进制消息发送到一些其他体系结构的机器上时你必须非常小心。一种简单的解决方案将所有消息编码成 ASCII（使用 `sprintf` 或 `pack`）码。本章描述的 RPC 模块使用了 `FreezeThaw` 库来获得网络透明的编码形式。

下面的代码描述了一个应用 `Msg` 模块的客户端：

```
use Msg;
$conn = Msg->connect( ocalhost 8080);
die "Error: Could not connect\n" unless $conn;
$conn->send_now("Message $i");
($msg, $err) = $conn->rcv_now();
```

`connect` 是一个创建连接对象（它的一个属性为套接字连接）的静态方法。`send_now` 方法把消息发送到那个连接上，而一个相应的名为 `rcv_now` 的方法将阻塞直到从另一方收到一个消息为止。我们马上就要看到延迟（或排队）的消息传递。

下面的代码描述了一个应用 `Msg` 的服务器：



```

use Msg;
use strict;
my $host = 'localhost';
my $port = 8080;
Msg->new_server($host, $port, \&login_proc);
print "Server created. Waiting for events";
Msg->event_loop();
#-----
sub login_proc {
    # 无条件的接由到来的连接请求
    return \&rcvd_msg_from_client;
}

sub rcvd_msg_from_client {
    my ($conn, $msg, $err) = @_;
    if (defined $msg) {
        print "$msg\n";
    }
}

```

这个脚本程序调用 `new_server` 来创建一个监听套接字（程序的地址），然后调用事件分发器 `event_loop`，这是位于 `select` 上的一层薄薄的包裹层。

当客户端企图连接时，`Msg` 就会创建一个局部连接对象，并以连接对象为参数调用你所提供的登录过程（为 `new_server` 提供的）。如果你想通过返回 `undef` 来拒绝一个连接请求，则可以在登录过程中，查询远端的主机地址和端口号。如果接受这个连接，那么就返回指向一个子例程的引用（在本例中就是 `rcvd_msg_from_client`），而对于继续在这个连接上收到的消息都将调用这个子例程。如果需要的话，不同的连接可以拥有不同的接收过程。

下面是你如何以延迟方式发送或接收消息的示例：

```

$conn = Msg->connect($remote_host, $remote_port, \&msg_from_server);
$conn->send_later($msg);
Msg->event_loop();

```

`Connect` 方法接受一个指向 `new_server` 这样的子例程的引用。`event_loop` 将在连接可写时，发送排队的外出消息，并负责将到来的消息分发到相应的本地子例程（延迟接收）。注意，如果一个客户端需要使用延迟消息，它必须调用 `event_loop`。

你发现“客户”与“服务器”之间的界限模糊起来了吗？它们都有事件循环（尽管客户端只有在延迟消息的情况下需要它），而且都要响应到来的消息。在传统的诸如数据库连接的客户服务器环境中，客户端初始一个连接并提出问题。服务器端从不会初始一个请求。在对等的环境中，譬如电话交谈，一个进程初始一个连接，但是一旦连接建立起来后，任何一方进程都可以发送消息。Msg支持这种对等模式。

其他的文件句柄可以像下面这样集成进来：

```
Msg->set_event_handler (\*STDIN, "read" => \&kbd_input);
```

这个进程现在就可以响应键盘事件了，而且还能兼顾到来的消息或在“后台”发送外出的排队消息。

所有事件驱动的应用框架都支持定时器时间来定期的触发一个后台任务。如果你有一个耗时的任务，你可以将它分成更易管理的片段并使用定时器（将超时值设置为0）来触发下一个子任务。这样你就可以不停的在每个子任务处理完毕后返回事件循环，而有机会处理其他不时到来的消息。由于这一章讲述的是有关网络计算的内容，我没有费心为Msg增添对定时器的支持。不过这只是小事一桩，因为select支持毫秒级的超时机制。

## Msg 的实现

Msg 对外提供公共接口如表 13-1 所示。

表 13-1 Msg 的公共接口

方法	描述
connect(host, port, [rcv_cb])	<p>连接到地址为 host 和 port 的服务器并返回一个连接对象。rcv_callback 为指向用户定义的子例程的引用，在远端进程发送一个消息时（任何时候），它将以如下形式被调用：</p> <pre>rcv_cb(\$conn, \$msg, \$err)</pre> <p>conn 为连接对象，用来发送消息和挂断连接。msg 为</p>

表 13-1 Msg 的公共接口 (续)

方法	描述
	接收到的消息；如果另一端关闭了连接，它的内容将为 <code>undef</code> （如果这种情况发生时， <code>Msg</code> 将会自动关闭这一端的连接）。如果有的话， <code>err</code> 包含最近一次 <code>sysread</code> 的错误值。
<code>\$conn-&gt;send_now(\$msg)</code>	立刻发送消息并在需要时阻塞。如果还有排队的消息，它会先在把它们发送出去之后再发送 <code>msg</code> 。
<code>\$conn-&gt;send_later(\$msg)</code>	把消息放到与连接对象关联的队列中并交由 <code>event_loop</code> 在套接字可写时在进行分发。就是说，你必须在某个时刻调用 <code>event_loop</code> ；否则消息永远不会被发送。
<code>\$conn-&gt;disconnect()</code>	关闭连接。
<code>(\$msg, \$err) = \$conn-&gt;rcv_now()</code>	阻塞直到接收到一个完整的消息为止。它并不调用提供给 <code>connect</code> 的回调函数。在标量变量的上下文中，它只返回消息；否则返回任何存在的错误代码。
<code>new_server(\$thishost, \$thisport, [login_proc])</code>	这是一个在 <code>thishost</code> 和 <code>thisport</code> 上创建监听套接字的静态方法。当远端的套接字企图 <code>connect</code> 时，将以连接对象和连接主机及端口为参数调用 <code>login_proc</code> 。如果 <code>login_proc</code> 返回 <code>undef</code> ，那么连接被关闭。
<code>set_event_handler( \$handle, ["read"=&gt; rd_cb], ["write"=&gt; wt_cb],</code>	<code>handle</code> 可以是套接字、文件、或管道句柄，也可以是从 <code>IO::Handle</code> 导出的对象。当相应的文件描述符准备好读写时， <code>event_loop</code> 将会调用回调函数。当回调值为 <code>undef</code> 时，已注册的回调将被删除。对于一个给定的句柄只能注册一个特定类型的回调函数。
<code>event_loop ([count])</code>	将给定的循环执行 <code>count</code> 次（默认情况为无限循环）。当为事件循环注册的句柄不存在时，循环将退出。请查看 RPC 中应用 <code>count</code> 的例子。

Msg 的实现被分成四个逻辑部分：

- 发送例程。用于连接到远端进程并向其发送消息。

- 接收例程。用于当消息或连接到来时接收通知。
- 非阻塞 I/O 的支持例程。在平台支持 POSIX 模块的情况下，将套接字设置为阻塞的或非阻塞的。
- 事件循环支持例程。用于分发与文件有关的事件。

让我们先来讨论发送端例程：

```
package Msg;
use strict;
use IO::Select;
use IO::Socket;
use Carp;
use vars qw(%rd_callbacks %wt_callbacks $rd_handles $wt_handles);
%rd_callbacks = ();
%wt_callbacks = ();
$rd_handles = IO::Select->new();
$wt_handles = IO::Select->new();
my $blocking_supported = 0;
```

## Msg: 发送端例程

```
sub connect {
    my ($pkg, $to_host, $to_port, $rcvd_notification_proc) = @_;
    # 创建一个新的网际套接字
    my $sock = IO::Socket::INET->new (
        PeerAddr => $to_host,
        PeerPort => $to_port,
        Proto    => 'tcp');

    return undef unless $sock;
    # 创建一个连接端对象
    my $conn = bless {
        sock => $sock,
        rcvd_notification_proc => $rcvd_notification_proc,
    }, $pkg;

    if ($rcvd_notification_proc) {
        # 将 _rcv 和 $conn 一起捆绑在一个闭包中
        my $callback = sub {_rcv($conn)};
        set_event_handler ($sock, "read" => $callback);
    }
}
```

```

    }
    $conn;
}

```

`connect` 建立一个客户端套接字并创建前面所提到的连接对象。连接对象是一个通信端并具有下列属性：

`sock`

socket 对象（属于类 `IO::Socket::INET` 类）。

`rcvd_notification_proc`

在接收到消息时调用的回调函数。

`queue`

指向缓冲消息列表的引用。

`send_offset`

在非阻塞模式下，`Msg` 允许部分写。如果套接字发生了阻塞，我们就记下已经为队列的最顶层消息发送了多少信息。

`msg`

在非阻塞模式下，`msg` 中包含了部分到达的消息。

`bytes_to_read`

包含了还需要读取的字节数。

一旦连接建立起来后，每一端都可以使用本地的连接对象来相互通话。

如果用户指定了一个回调函数（`$rcvd_notification_proc`），我们就设置我们的事件处理程序来调用一个私有例程 `_rcv`，它将会在一个完整的消息被接收到以后接着调用这个回调函数：

```

sub disconnect {
    my $conn = shift;
    my $sock = delete $conn->{sock};
    return unless defined($sock);
    set_event_handler ($sock, "read" => undef, "write" => undef);
    close($sock); undef $!; # 理想状态下应当处理来自 close 的错误信息
}

```

```

sub send_now {
    my ($conn, $msg) = @_;
    _enqueue ($conn, $msg);
    $conn->_send (1); # 1 ==> flush
}

```

`send_now`将消息入队，并告诉`_send`去刷新此消息以及其他先前逗留在队列中的消息。

```

sub send_later {
    my ($conn, $msg) = @_;
    _enqueue($conn, $msg);
    my $sock = $conn->{sock};
    return unless defined($sock);
    set_event_handler ($sock, "write" => sub {$conn->_send(0)});
}

```

`send_later`将消息入队并注册一个执行“写”操作的回调函数。它将在以后调用`event_loop`且文件描述符可写时被调用。

```

sub _enqueue {
    my ($conn, $msg) = @_;
    # prepend length (encoded as network long)
    my $len = length($msg);
    $msg = pack ('N', $len) . $msg;
    push (@{$conn->{queue}}, $msg);
}

```

`_enqueue`在每个消息前面缀上长度信息，并推送到与连接对象关联的队列中。长度信息编码为“独立于网络的长整型”（一个32位的数字），因此接收端能够准确的知道读取4个字节来获得长度信息。我们前面已经提到过，消息本身是不用考虑字节顺序问题的。

```

sub _send {
    my ($conn, $flush) = @_;
    my $sock = $conn->{sock};
    return unless defined($sock);
    my ($rq) = $conn->{queue}; # rq -> ref. to queue.

    # 如果设置了 $flush, 那么将套接字设置为阻塞模式, 并将所有
    # 队列中的消息发送出去, 只在发生错误时返回

```

```

# 如果 $flush 设置为 0 (延迟模式) 将 socket 套接字设置为非阻塞模式,
# 只在每个消息发送后或有可能在发送消息当中阻塞时, 才返回事件循环

$flush ? $conn->set_blocking() : $conn->set_non_blocking();
my $offset = (exists $conn->{send_offset}) ? $conn->{send_offset} : 0;
while (@$rq) {
    my $msg          = $rq->[0];
    my $bytes_to_write = length($msg) - $offset;
    my $bytes_written = 0;
    while ($bytes_to_write) {
        $bytes_written = syswrite ($sock, $msg,
                                    $bytes_to_write, $offset);
        if (!defined($bytes_written)) {
            if (_err_will_block($!)) {
                # 应当只发生在延迟模式
                # 记下我们已经发送的数量
                $conn->{send_offset} = $offset;
                # 由于事件处理程序应当已经进行了设置
                # 因此我们最终要进行回调并恢复发送
                return 1;
            } else { # Uh, oh
                $conn->handle_send_err($!);
                return 0; # 失败了, 消息仍保留在队列中……
            }
        }
        $offset += $bytes_written;
        $bytes_to_write -= $bytes_written;
    }
    delete $conn->{send_offset};
    $offset = 0;
    shift @$rq;
    last unless $flush; # 返回 select 等候下一个操作事件的到来
}
# 如果队列还没有完, 就回调我
if (@$rq) {
    set_event_handler ($sock, "write" => sub {$conn->_send(0)});
} else {
    set_event_handler ($sock, "write" => undef);
}
1; # 成功
}

```

send 完成实际的消息发送工作, 它或是由 send\_now 直接调用或是作为事件循环的回调函数被调用。如果是从 send\_now 中调用, 它会将套接字设置为阻塞模

式并刷新队列中的所有消息。如果是在事件循环中被调用，它会将套接字设置为非阻塞模式，并在返回事件循环以前每次最多刷新队列中的一条消息。这样别的连接也能获得自己的运行时间。如果 `syswrite` 表示要阻塞了，`_send` 将会记下这条消息已经发送了多少（在 `send_offset` 属性中）并返回事件循环。无论何种情况，它都要考虑到 `syswrite` 或许只写了一部分缓冲区这一事实。

```
sub handle_send_err {
    # 要想更好的处理发送错误，就从 Msg 导出相应的子类并重新 bless $conn
    my ($conn, $err_msg) = @_;
    warn "Error while sending: $err_msg \n";
    set_event_handler ($conn->{sock}, "write" => undef);
}
```

这是一个空泛的错误处理过程，它除了关掉事件通知以外别的什么都不做。它不对连接对象做任何操作，因此你能够潜在的从停下的地方继续进行处理。要完成这项工作，你必须在导出类中重载这个方法（请看后面有关 RPC 模块中的例子）。

## Msg：接收端例程

这一节中的程序实现了监听端的功能：

```
my ($g_login_proc, $g_pkg); # 前缀 g_ 表示 global
my $main_socket = 0;
sub new_server {
    @_ == 4 || die "new_server (myhost, myport, login_proc)\n";
    my ($pkg, $my_host, $my_port, $login_proc) = @_;

    $main_socket = IO::Socket::INET->new (
                                                LocalAddr => $my_host,
                                                LocalPort => $my_port,
                                                Listen    => 5,
                                                Proto       => 'tcp',
                                                Reuse       => 1);
    die "Could not create socket: $! \n" unless $main_socket;
    set_event_handler ($main_socket, "read" => \&_new_client);
    $g_login_proc = $login_proc; $g_pkg = $pkg;
}
```

`new_server` 与 `connect` 多少有点相似。它创建一个监听套接字并向事件处理程序



注册一个用户定义的登录过程。(那些不愿发送或接收延迟消息的客户端程序不必调用 `new_server` 或 `event_loop`。)该登录过程只有在服务器调用 `event_loop` 并且有连接请求到来时才会被调用。与 `connect` 不同, `new_server` 并不创建一个连接对象; 这是 `_new_client` 的工作:

```
sub _new_client {
    my $sock = $main_socket->accept();
    my $conn = bless {
        'sock' => $sock,
        'state' => 'connected'    }, $g_pkg;
    my $rcvd_notification_proc = &$g_login_proc ($conn);
    if ($rcvd_notification_proc) {
        $conn->{rcvd_notification_proc} = $rcvd_notification_proc;
        my $callback = sub {_rcv($conn)};
        set_event_handler ($sock, "read" => $callback);
    } else { # Login failed
        $conn->disconnect();
    }
}
```

当收到一个连接请求时, `_new_client` 被调用。在执行完 `accept` 后, 它会给用户定义的登录过程一次接受或拒绝该连接的机会。这个代码引用立即与新建的连接对象相关联, 并在消息到达那个连接时被调用。`_rcv` 注册为标准的回调函数, 由它来处理所有到来的消息(对于所有的连接), 并在积攒了一条完整的消息之后才调用上面提到的代码引用。

```
sub _rcv {
    my ($conn, $rcv_now) = @_;
    # 如果有话, 就查出已经接收了多少
    my ($msg, $offset, $bytes_to_read, $bytes_read);
    my $sock = $conn->{sock};
    return unless defined($sock);
    if (exists $conn->{msg}) {
        $msg = $conn->{msg};
        delete $conn->{'msg'};
        $offset = length($msg);
        $bytes_to_read = $conn->{bytes_to_read};
    } else {
        # 典型的情况 ...
        $msg = "";
    }
    # 对 _send 的补充
    # $rcv_now 为 $flush 的补充
    # 已经创建了一个拷贝
    # sysread 将其追加到它后面
    # 不然 -w 会报错
```

```

        $offset          = 0 ;
        $bytes_to_read = 0 ;                                # 马上就要进行设置
    )
    # 我们需要以阻塞模式读取到消息的长度信息
    # 读取 4 个字节而发生长时间阻塞的可能性很小
    if (!$bytes_to_read) {                                    # 获取新的长度信息
        my $buf;
        $conn->set_blocking();
        $bytes_read = sysread($sock, $buf, 4);
        if ($! || ($bytes_read != 4)) {
            goto FINISH;
        }
        $bytes_to_read = unpack ('N', $buf);
    }
    $conn->set_non_blocking() unless $rcv_now;
    while ($bytes_to_read) {
        $bytes_read = sysread ($sock, $msg, $bytes_to_read, $offset);
        if (defined ($bytes_read)) {
            if ($bytes_read == 0) {
                last;
            }
            $bytes_to_read -= $bytes_read;
            $offset        += $bytes_read;
        } else {
            if (_err_will_block($!)) {
                # 只能以非阻塞模式运行到这里
                $conn->{msg}          = $msg;
                $conn->{bytes_to_read} = $bytes_to_read;
                return ; # ... 返回事件循环
                        # 当套接字再次可读时 _rcv 将会被调用
            } else {
                last;
            }
        }
    }
    # 消息读取成功
FINISH:
    if (length($msg) == 0) {
        $conn->disconnect();
    }
    if ($rcv_now) {
        return ($msg, $!);
    } else {
        &{$conn->{rcvd_notification_proc}}($conn, $msg, $!);
    }

```

```

    }
}

```

`_rcv`是对 `send`的补充,它完成从套接字中读取数据这种枯燥的工作。与 `_send`不同,它并不知道自己要处理多少数据,但却知道任何消息的前四个字节包含有编码后的长度信息(余下的信息的长度)。为使问题简单化,它在读取这四个字节以前先设置为阻塞模式,并寄(合理的)希望于,即便发生了阻塞,也不会时间太久。一旦长度信息被解码,它又会在需要时把模式设置回非阻塞模式,并继续从套接字中读数据。和 `_send`一样,它也考虑到 `sysread`可能返回比所要求的少的数据,或者会返回表示它要阻塞的错误信息。如果套接字要阻塞了,那么 `_rcv`就会把不完整的消息拷贝到连接对象中,记下还需要读取的字节数然后返回,等待事件循环的下一次调用。如果出现了错误,它就会自动的中断连接。

```

sub rcv_now {
    my ($conn) = @_;
    my ($msg, $err) = _rcv ($conn, 1); # 1 表示立即接收
    return wantarray ? ($msg, $err) : $msg;
}

```

## Msg: 对非阻塞 I/O 的支持

```

BEGIN {
    eval {
        require POSIX; POSIX->import(qw(F_SETFL O_NONBLOCK EAGAIN));
    };
    $blocking_supported = 1 unless $@;
}

```

`BEGIN`用于测试它是否能够加载POSIX模块,如果能行,它就会设置 `$blocking_supported`,而 `$blocking_supported`则由下面的例程使用:

```

sub _err_will_block {
    if ($blocking_supported) {
        return ($_[0] == EAGAIN());
    }
    return 0;
}

sub set_non_blocking {
    if ($blocking_supported) {

```

```

        # 保留其他的 fcntl 标志位
        my $flags = fcntl ($_[0], F_GETFL(), 0);
        my $conn = shift;
        fcntl ($conn->{sock}, F_SETFL(), $flags | O_NONBLOCK());
    }
}

sub set_blocking {
    if ($blocking_supported) {
        my $flags = fcntl ($_[0], F_GETFL(), 0);
        $flags &= ~O_NONBLOCK(); # 清除阻塞标志, 但保留其余的标志
        my $conn = shift;
        fcntl ($conn->{sock}, F_SETFL(), $flags);
    }
}

```

我们将在最后一章讲到, `set_blocking` 与 `set_non_blocking` 都要调用 `fcntl`。`F_SETFL` 将文件描述符的标志位设置成你所提供的位掩码。因此我们一定要小心不要清除或许已经设置了的标志位。

### Msg: 事件循环例程

事件循环支持例程使用 `IO::Select` 来管理文件句柄和套接字句柄集。我们前面描述的发送和接收端例程就调用这些例程。但是由于这些程序从不假设是谁在调用它们, 因此它们处在较低的逻辑层。这也就意味着, 为了使这个模块同其他事件驱动的工具包能够共存, 你将只需重写下面的这些例程 (但仍然保留原来的接口)。例如, 为了使 `Msg` 能够与 `Tk` 一起工作, 你可以让 `set_event_handler` (如下所示) 简单的将其功能移交给等价的名为 `fileevent` (将在第十四章“使用 `Tk` 进行用户界面编程”中讲述) 的过程来完成; 同样, `event_loop` 可以简单的调用 `Tk` 的 `run` 方法, 而不是调用 `IO::Select`。

```

sub set_event_handler {
    shift unless ref($_[0]); # 如果第一个参数为包名就执行 shift
    my ($handle, %args) = @_;
    my $callback;
    if (exists $args{'write'}) {
        $callback = $args{'write'};
        if ($callback) {
            $wt_callbacks{$handle} = $callback;
            $wt_handles->add($handle);
        }
    }
}

```

```

    } else {
        delete $wt_callbacks{$handle};
        $wt_handles->remove($handle);
    }
}
if (exists $args{'read'}) {
    $callback = $args{'read'};
    if ($callback) {
        $rd_callbacks{$handle} = $callback;
        $rd_handles->add($handle);
    } else {
        delete $rd_callbacks{$handle};
        $rd_handles->remove($handle);
    }
}
}
}

```

set\_event\_handler通过以句柄为散列表索引来记录读和写回调函数。你可以使用回调值为undef的set\_event\_handler来删除一个回调：

```

sub event_loop {
    my ($pkg, $loop_count) = @_;
    my ($conn, $r, $w, $rset, $wset);
    while (1) {
        # 如果没有剩余的句柄需要处理就退出循环
        last unless ($rd_handles->count() || $wt_handles->count());
        ($rset, $wset) =
            IO::Select->select ($rd_handles, $wt_handles,
                                undef, undef);
        foreach $r (@$rset) {
            &{$rd_callbacks{$r}} ($r) if exists $rd_callbacks{$r};
        }
        foreach $w (@$wset) {
            &{$wt_callbacks{$w}} ($w) if exists $wt_callbacks{$w};
        }
        if (defined($loop_count)) {
            last unless --$loop_count;
        }
    }
}

```

event\_loop通常为无限循环，但是可以指示它来完成限定次数的循环。这种指定

循环次数的思想能够分发其他的事件而不会丧失对无限循环的控制。查看一下下一节将要描述的 RPC 的实现，它使计数器为 1 而以一种可控的方式分发事件。

## 远程过程调用 (RPC)

在这一节，我们使用 `Msg` 库来实现远程过程调用模块，`RPC.pm`。RPC 的思想就是透明的调用另一个进程空间中的子例程，并使它看起来就像在自己的进程中调用一样。下面列出的是我们认为调用普通过程时所应具备的功能，也是 RPC 模块需要考虑的问题：

### 同步性

调用者将一直等到被调用过程执行完毕。RPC 模块通过调用 `Msg::send_now` 和 `Msg::rcv_now` 来获得这种阻塞式的行为。

### 参数

Perl 的子例程可以接受任意数量任意类型的参数（包括指向对象，复杂数据结构和子例程的引用）。RPC 模块通过使用我们在第十章“持续性”中所讲的 `FreezeThaw` 模块来对参数进行编组：所有的参数都被展平并编码成单一的字符串（冻结）并在另一端进行恢复（融化）。这就意味着所有以引用方式发送的数据结构都需要进行整体的拷贝，这样接收端的子例程才能通过引用访问到所指向的对象（就像在同一个进程中的情形那样）。`FreezeThaw`——继而是 RPC，都不考虑对代码的引用，因为没有办法（在 Perl 中）来解码一个代码引用并获取子例程的程序代码（因为它可能被编译成了机器代码）。我们可以在远端创建一个哑过程，并让它来执行嵌套的 RPC 调用来调用真正的代码引用，但是当前的实现并没有这项功能（但并不排除实现的可能性）。

### 上下文环境

一个子例程可以通过使用 `wantarray` 来查明调用者指定的上下文是列表还是标量变量。一个子例程被从远端进程来调用不成问题。RPC 模块来管理这种所需的透明性。另一个上下文（不要与 Perl 中 `context` 的意思相混淆了）的例子是调用者的包。当你使用 `foo()` 时，你就是指当前包中的 `foo()`。

### 例外

一个子例程可以调用die并期望它的调用者来捕获它。RPC的接收端在eval中调用目标子例程，并在它退出时将消息发送回调用进程，而调用者进程将会在自己的空间中调用die来捕获这个错误。

### 相互递归

子例程A可以调用子例程B，而B则又可以继续调用A——这被称做相互递归（mutual recursion）。RPC允许这种情况，因为它能够在send上阻塞时，也能处理到来的消息。

### 没有死锁

传统的RPC系统在两个对等的进程同时调用对方时容易导致死锁，因为如我们在第十二章所看到的，它们太不礼貌，谁也不想听对方在说些什么。这对于RPC来说不成问题。实际上，尽管从调用者的观点来看，它处在阻塞状态，但是它仍旧能够分发从所有文件描述符到来的消息。

### 没有代码生成

典型的RPC系统要生成客户和服务存根代码（stub code），但是RPC不需要这样。这应归功于Perl是一种动态语言。

## 使用 RPC

让我们来看一个RPC模块的应用范例。先描述的是客户端的代码：

```
# 客户端的内容
use RPC;
my $conn = RPC->connect($host, $port);
my $answer = $conn->rpc('ask_sheep',
                        "Ba ba black sheep, have you any wool ?");
print "$answer\n";
```

给定主机和端口号，客户端将建立一个RPC连接。子例程一般来说按如下方式调用：

```
$answer = ask_sheep ($question);
```

使用 RPC 则这样调用:

```
$answer = $conn->rpc ("ask_sheep", $question);
```

客户端代码知道它在执行一个RPC调用。要使这种操作变得透明实在是相当的简单。使用 eval, 我们可以在调用者一端动态的创建一个名为 ask\_sheep 的哑客户存根, 并让它来调用 rpc()。

然而被调用的子例程不会知道它是本地调用还是远程调用 (当然除非它使用 caller() 来弄清处)。

远端的进程 (称之为 RPC 服务器) 提供所需的子例程, 并调用 new\_server 和 event\_loop 来接收到来的 RPC 调用; ask\_sheep 将会被很好的调用。简单!

```
# 服务器端的内容
RPC->new_server($host, $port);
RPC->event_loop();

sub ask_sheep { # 从客户端调用的样板子例程
    print "Question: @_\\n";
    return "No";
}
```

现在, 让我们来看一个在对等进程间使用RPC的例子。进程1 (由 \$host1 和 \$port1 标识) 调用进程2 (\$host2, \$port2) 的子例程 two, 而它又会调用进程1的子例程 one。

进程1如下:

```
sub one {
    print "One called\\n";
}
$conn2 = RPC->new_server($host2, $port2);
$conn2->rpc ("two");
```

进程2如下:





```

sub two {
    print "Two called\n";
}
$conn1 = RPC->new_rpc_server($host1, $port1);
$conn1->rpc ("one");

```

每个进程都调用 `new_rpc_server` 来建立一个监听端口。由于 `rpc` 调用在监听到来消息的同时还能发送消息，所以它们两个进程都不必显式的调用 `event_loop`。一个打算空闲一段时间的进程当然可以这么做。

## RPC：实现

RPC 的实现令人吃惊的短小，我们应当感谢 `Msg` 和 `FreezeThaw` 模块。它继承 `Msg` 来提供同样的连接和事件循环抽象。

让我们先来检查一下调用端：

```

package RPC;
use Msg;
use strict;
use Carp;
@RPC::ISA = qw(Msg);
use FreezeThaw qw(freeze thaw);

sub connect {
    my ($pkg, $host, $port) = @_;
    my $conn = $pkg->SUPER::connect($host, $port, \&_incoming_msg);
    return $conn;
}

```

`connect` 只是简单的调用 `Msg` 的 `connect`，并将 `_incoming_msg` 作为在所有消息（包括对子例程调用的响应和文件终止指示）到来时的通知子例程。它交由 `Msg` 的 `connect` 来创建连接对象并在 `RPC` 下 `bless`。`Msg` 和 `RPC` 按照可以被继承的方式编写；包名不是硬编码的。

```

my $g_msg_id = 0;
my $send_err = 0;
sub handle_send_err {
    $send_err = $!;
}

```

```
}
```

handle\_send\_err 重载了 `Msg::handle_send_err`, 并将碰到的任何错误保存起来。如下所示, 该错误代码在 `rpc` 中进行检查。RPC 与 `Msg` 中的错误处理远未达到标准, 而且要想在生产应用中可靠的使用之前, 还需要有相当的工作要做。

```
sub rpc {
    my $conn = shift;
    my $subname = shift;

    $subname = (caller() . '::' . $subname) unless $subname =~ /:/;
    my $gimme = wantarray ? 'a' : 's'; # 数组或标量变量
    my $msg_id = ++$g_msg_id;
    my $serialized_msg = freeze ('>', $msg_id, $gimme, @_);
    # 发送与接收
    $conn->send_later ($serialized_msg);
    do {
        Msg->event_loop(1); # 分发其他消息直到获得一个应答为止
    } until (exists $conn->{rcvd}->{$msg_id} || $send_err);
    if ($send_err) {
        die "RPC Error: $send_err";
    }

    # 消息出队
    my $rl_retargs = delete $conn->{rcvd}->{$msg_id}; # 指向列表的引用
    if (ref($rl_retargs->[0]) eq 'RPC::Error') {
        die "${rl_retargs->[0]}";
    }
    wantarray ? @$rl_retargs : $rl_retargs->[0];
}
```

`rpc` 使用 `FreezeThaw` 模块的 `freeze` 方法来将下面的信息捆绑到一个大的字符串中:

- 远程子例程的名字。如果子例程的名字不是全限定的, 那么调用者模块将会缀到它的前面, 这也是普通子例程所期望的。
- 子例程的参数。
- `wantarray` 指示 (`$gimme`): “s” 表示 scalar, “a” 表示数组。

- 请求或应答指示。“>”指示请求，而“<”指示应答。每当接收者获得一个消息，就应该知道它是一个对外出消息的应答呢还是一个期望进行计算的到来请求。
- 消息标识。它被用来标识与请求相对应的应答。

freeze方法考虑到了循环数据结构和对象并返回一个ASCII字符串，这就意味着我们不用担心本地的整数或双精度浮点数的尺寸，以及它们在内存中的排列情况（字节顺序）。我们使用了Msg->send\_later()，这是因为它尽可能的触发非阻塞I/O。只有当event\_loop被调用时，消息才真正被发送出去，因为要由它来判定套接字什么时候才可写。与此同时，event\_loop来跟踪其他到来的消息并把它们分发出去。计数器为1将迫使事件循环在分发完一个回合的消息后立刻返回，这样我们仍可以获得控制。当远端机器的响应到来时，event\_loop将调用\_incoming\_msgs，由它对响应进行解码，并将返回值挂接到连接对象上。请接着向下读。

现在让我们来看一下接收端：

```
sub new_server {
    my ($pkg, $my_host, $my_port) = @_;
    $pkg->SUPER::new_server($my_host, $my_port,
                           sub {$pkg->_login(@_)});
}
sub _login {
    \&_incoming_msg;
}
```

new\_server 与 connect 一样是对 Msg 相应函数的简单包裹层。默认情况下，所有的接入连接均被无条件的接受，并且消息被转给子例程\_incoming\_msg，如下所示。通过 \$pkg 间接的调用\_login模块，可以使你通过继承 RPC 来提供自己的\_login过程，并在需要时拒绝连接。

```
sub _incoming_msg {
    my ($conn, $msg, $err) = @_;
    return if ($err); # 需要更好的错误处理
    return unless defined($msg);
    my ($dir, $id, @args) = thaw ($msg);
```

```

my ($result, @results);
if ($dir eq '>'){
    # 新的请求消息
    my $gimme = shift @args;
    my $sub_name = shift @args;
    eval {
        no strict 'refs'; # 因为我们使用符号引用来调用这个子例程
        if ($gimme eq 'a') { # 希望返回一个数组
            @results = &{$sub_name} (@args);
        } else {
            $result = &{$sub_name} (@args);
        }
    };
    if ($?) {
        $msg = bless \$_, "RPC::Error";
        $msg = freeze('<', $id, $msg);
    } elsif ($gimme eq 'a') {
        $msg = freeze('<', $id, @results);
    } else {
        $msg = freeze('<', $id, $result);
    }
    $conn->send_later($msg);
} else {
    # 响应我们以前发送的消息
    $conn->{rcvd}->{$id} = \@args;
}
}

```

`_incoming _msg` 对应于 `rpc` 方法。它打开由 `rpc` 发送的消息并检查其发送方向（是请求还是响应）。如果是一个请求，它通过符号引用来调用所需的子例程。注意，根据 `wantarray` 的指示，它提供了标量变量或向量数组形式的结果。如果 `eval` 报告了错误，那么变量 `$_` 将被填上一个 `RPC::Error` 的错误标签并返送回调进程（这将会引发 `die`）。

## 相关资源

下面的资源提供了有关消息传递和 `RPC` 的有用信息：

1. `EventServer`。

可以从 CPAN 获得，提供了一个对 Msg 中的 event\_loop 过程的替代品，并支持对文件和超时事件的回调。

2. Unix Network Programming. W. Richard Stevens. Prentice-Hall, 1990.
3. Advanced Programming in the Unix Environment. W. Richard Stevens. Prentice-Hall, 1992.
4. “A Note on Distributed Computing.” Jim Waldo, Geoff Wyant, Ann Wollrath, 和 Sam Kendall, 获取地址为 <http://www.sunlabs.com/techrep/1994/abstract-29.html> (Technical Report TR-94-29)。

一份写得相当出色的与 RPC（概念）和一般分布式计算有关的问题的报告（实际上 Sun 研究实验室的其他技术报告也是很不错的，值得一读）。



# 第十四章

## 使用 Tk 进行用户 界面编程

本章简介：

- GUI、Tk 和 Perl/Tk 简介
- 开始使用 Perl/Tk
- 组件之旅
- 布局管理
- 定时器
- 事件联编
- 事件循环
- 相关资源

X Window 系统编程就像使用罗马数字  
来算 PI 的平方根一样。

—— 无名氏

在本章，我们要学习如何使用功能最为丰富和外观专业化的Tk工具箱来创建用户界面。我们开始将简单浏览一下大多数的Tk组件以及一部分Tix扩展组件，然后再学习布局管理（geometry management）（如何在表单上安排组件）。接着，我们将简单的查看一下Perl对定时器的支持，它会在第十五章“GUI实例：Tetris”中有大量应用。我们然后将讨论事件联编，它可以让我们把鼠标和键盘的事件任意组合映射到回调过程中。最后我们要讨论事件循环的问题，这与我们在第十二章“使用套接字进行网络编程”中的内容有些相似。

为了简单起见，这一章将提供一些相互之间没有什么联系的代码片段，来演示各种组件和Tk的其他特性；我们将在下面的两章中以大量的篇幅，来讲述所有这些方面，并在实际问题中加以运用。

虽然我们这一章谈论的主题是创建用户界面，请你还是最好能读一下Alan Cooper（注1）优秀而且极有见地的著作：《About Face: The Essential of User Interface Design》。

---

注1： 他被称为“Visual Basic之父”。

## 对 GUI, Tk 和 Perl/Tk 的介绍

从本质上说,所有的窗口平台(包括苹果公司的 Macintosh、X Window 和微软的 Windows)都很简单。它们提供一种低层 API 来创建和管理窗口,报告诸如鼠标和键盘等令人感兴趣的事件,绘制诸如线、圆和位图等图形元素。问题就在于,即便是创建一个简单的窗体,也要书写相当数量的代码,阅读几千页的文档。

那些常用模式的 GUI 代码逐渐演变成了组件(widget)[在微软 Windows 平台中则称做“控件(control)”,例如按钮、滚动条和列表等。现在创建一个 GUI 应用是一项简单的工作,你只需执行一个交互式窗体设计器,并将预先制作好的部件拖放到窗体上并按照你的意愿进行摆放。面向对象编程从没有这样简单过。

因此,组件与脚本语言是一对最佳拍档。组件拥有简单的界面,而基于窗体的图形用户界面对性能的要求又不是关键因素。这两个特点使得处理图形用户界面很适合脚本编程。同时 GUI 又需要更高的可定制性(因为这是同用户打交道的应用部分,而且一般来说 GUI 就是应用系统),这样你就能够理解诸如 Visual Basic、PowerBuilder 和 Hypercard 等工具广受欢迎的原因了。

在 Unix 系统中, X Window 一直是窗口平台的选择。基于 X 已经建立了好几个组件工具箱,如: Athena、InterView、Motif 和 Tk。从专业化外观、易用性和文档上来说, Tk 更胜一筹。最为要紧的是,它是自由软件。

和其他的组件工具箱不同, Tk 是特意为脚本语言 Tcl 而开发的(注 2)。确实可以说 Tk 是 Tcl 得以普及的主要原因。有不少人都不喜欢作为脚本语言的 Tcl 但却喜欢 Tk, 而且试图将它改制到自己喜爱的脚本语言中——如 Scheme、Python、Guile, 当然还有 Perl。Malcolm Beattie 做了最初的尝试,他通过在内部使用 Tcl 解释器来存取 Tk 库的方式,提供了一个 Perl 代码层。

Nick Ing-Simmons 尝试了一种更为费力的方法:他删除了 Tk 中所有内嵌的 Tcl 代码,并为它编制了一个通用可移植层,以使它可以方便的加入其他脚本语言中。这项工作被称做 pTk(可移植 Tk)。他为这一层代码增加了 Perl5 的包裹层(目的

---

注 2: Tcl 与 Tk 都是由当时在加州大学伯克利分校的 John Ousterhout 博士开发的,他现在供职于 Sun 公司。请见 <http://www.sunlabs.com/research/Tcl>。

是为了将来为其他语言增加包裹层)。pTk同Perl包裹模块*Tk.pm*的组合被称之为Perl/Tk, 也就是本章的主题。

与此同时, Sun公司Ousterhout博士的小组将Tcl和Tk移植到Microsoft Windows和Mac上, 而Perl/Tk组合也照样紧跟其后。其他可移植的GUI选项有Tcl/Tk, 和Python/Tk (它不依赖于pTk)。微软正在将它的ActiveX(以前称做OLE)和VBA(Visual Basic for Application)移植到Unix环境中, 因此它们或许很快就会成为可怕的竞争者。VB工具箱从功能上来说远不及Perl和Tk, 但是它的开发环境和第三方支持是难以战胜的。我们生活在一个有趣的时代!

有几个新的由Ioi Kim Lam开发的组件, 具有专业化外观, 名为Tix, 已经以扩展库的形式加入到了Tk中。其中包括气球提示(用于显示帮助信息)、记事本和电子表格形式的栅格组件(grid widget)。有个好消息, 那就是Perl/Tk的发行版中, 也包含了建立在它上面的Perl代码层。

## 开始使用 Perl/Tk

所有使用Perl/Tk开发的用户界面都遵循下面的总体顺序:

1. 创建一个主窗口(也被称做顶层窗口)。
2. 实例化一个或多个组件, 进行配置, 然后把它们安放在主窗口中。一个组件就是一组产生可视界面元素的数据与方法的集合, 如按钮和列表框, 并可以让它在点击或其他操作下做出相应的动作。
3. 启动事件循环。启动后, 用户的动作(事件)将决定程序做些什么。

例14-1描述了这些以串行方式执行的步骤; 它们最终产生如图14-1所示的图形用户界面(注3)。

例14-1: 简单的用户界面代码

```
use Tk                                     # 加载模块
#-----
```

注3: 尽管它没有任何交互性——实际上也没有多少图形化的内容。



```

# 创建主窗口
#-----
$stop = MainWindow->new();
$stop->title ("Simple");
#-----
# 实例化组件并加以排列
#-----
$l1 = $stop->Label(text => 'hello',          # 标签属性
                  anchor => 'n',            # 将文本排放到“北方”
                  relief => 'groove',       # 边界风格
                  width => 10, height => 3); # 10 个字符宽, 3 个字符高
$l1->pack();      # 在主窗口中为它设定默认的位置
#-----
# 进入无限循环以分发到来的事件
#-----
MainLoop();

```

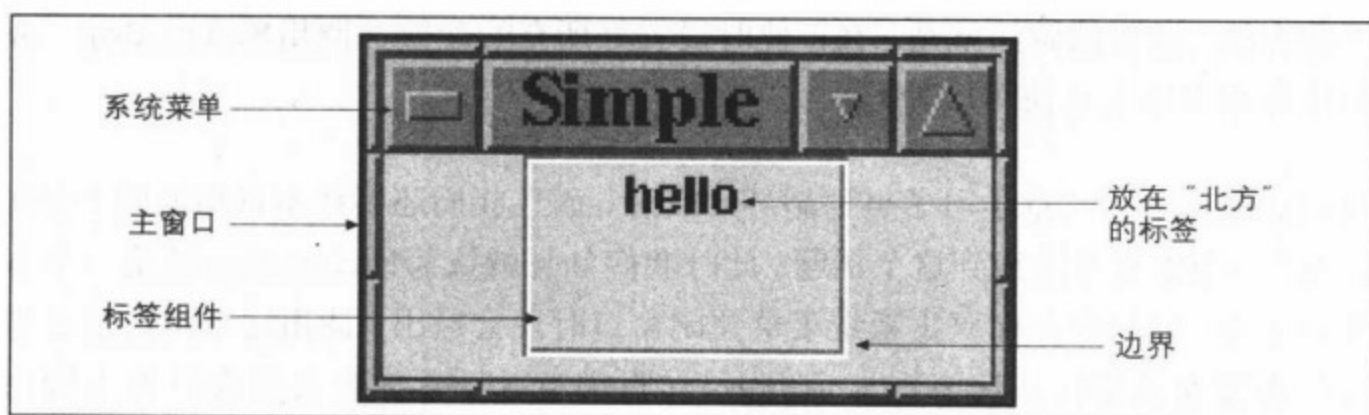


图 14-1 我们第一个 Perl/Tk 程序的屏幕显示

这个例子涵盖了许多重要的 Tk 概念（和大多数 GUI 工具包的整体概念）所包围。

主窗口是最外层的外壳，它由缩放手柄（resize handle）、系统菜单和最大与最小化框（也被称做修饰物）。一个应用程序可以有多个主窗口。

然后我们让主窗口来创建一个带有预设属性的标签（label）组件。你可以通过调用它的 `configure` 方法来改变组件的属性：

```
$label->configure (text => 'foobar', foreground => 'red');
```

有一些组件，如 `Frame` 和记事本（`Notebook`），自身还能包含其他的组件，因此组件的层次可以任意的进行嵌套。主窗口总是位于层次的根部。

接下来我们调用了组件的 `pack` 方法来进行布局管理：也就是设置组件的位置、高度、宽度等样式。这个调用只是简单的移交给组件的包容器来完成，这里是指主窗口，由它来计算屏幕的实际面积并为所包含的组件分派摆放空间。这类似于在说“sock->pack”，并让手提箱计算出在哪里存放这些袜子，以及排放一起需要多大的空间。

Packing只是众多用于摆放组件的布局管理模式中的一种。Tk提供了`grid`和`placer`布局管理器。我们将在本章的“布局管理”一节学到。

你也可以像下面这样将创建与摆放一次完成：

```
$l = $top->Label (text => 'Oh my')->pack();
```

在大多数情况下，你甚至不必捕获它的返回值，除非你计划以后还要调用那个组件的方法。通常的做法就是，在创建时设置好所有的参数并调用`MainLoop`。我们将在本书中大量使用这种方式。

我们已经在第十二章学习了事件循环的概念，而且我们还要在本章后面的“事件循环”一节中更多地谈到这个话题。此时我们姑且就认为`MainLoop`就是一个事件分发器，它只有在你双击系统菜单关闭窗口时才会退出。调用这个函数是必需的；否则你的窗体永远不会显示出来。（附带说一下，你也必须在组件上调用`pack`；否则该组件也无法显示出来。）

要讲的就这么多。现在我们只需知道都有哪些组件可用，它们都支持什么样的属性，还有如何将它们联编起来。请接着读下去！

## GUI 窗体：一种简单的方式

如你可以将它们画出来，为什么还要编写代码来创建静态的屏幕显示呢？Sun公司Tcl/Tk小组的Stephen Uhler曾编写了一个所见即所得的GUI构造器，名叫SpecTcl（发音为“spectacle”），并打算支持多种语言。这个工具被定制为支持Perl/Tk、Java/Tk和Python/Tk；相应的软件包分别叫做SpecPerl、SpecJava和

SpecPython。Mark Kvale 完成了 Perl/Tk 的移植工作，这个软件包可以从他的主页获得（注 4）：<http://www.keck.ucsf.edu/~kvale/specPerl/>。

使用 SpecPerl，你能够可视化地摆放组件，在相应的窗体中设置组件相关的属性，以及从调色板中选择颜色和字体——非常方便。

但是我们在这一章（还有后面的两章）将手工编写 GUI 代码，而不是使用 SpecPerl。这里有几个原因。首先我们创建的并不是复杂的窗体。其次大多数的例子重点放在研究 Tk 的动态性上，而使用 GUI 构造器只能帮助你创建静态的窗体。第三就是一旦你理解了本章所讲的内容，你就会知道 SpecPerl 都生成了哪些代码。

## 组件之旅

这一节提供了对大多数 Tk 和 Tix 中所实现的有趣的组件类的介绍，并练习使用常用的配置选项和方法。为了减少混乱并为今后使用提供快速的查找功能（当你知道要找些什么时），在附录一“Tk 组件参考”中提供了属性和方法的集合。你要清楚的一点是，尽管本章信息量大，它还只是整个 Tk 组件功能集的一个子集。它并没有覆盖由 Tk 和 Tix 所提供的所有组件。Perl/Tk 软件包中包含了完整的写得很好而且全面的 Tk 文档。

## 组件属性

请浏览一下附录一中的表 A-1，大致了解一下所有组件所共有的可配置的各类属性。这些属性大多数为字符串和数字，但是在我们继续讨论实际的组件之前，有必要先来深入研究一下三种类型的属性：字体、图片和颜色。

### 字体

字体值以 XLFD(X Logical Font Description)的格式来描述，这种格式由 14 个以连字符分隔的字段组成，如图 14-2 所示。

---

注 4：Sun 从那以后开始商业化的销售 SpecTcl，因此 SpecTcl 必然基于过时的（而且是免费的）SpecTcl 代码。

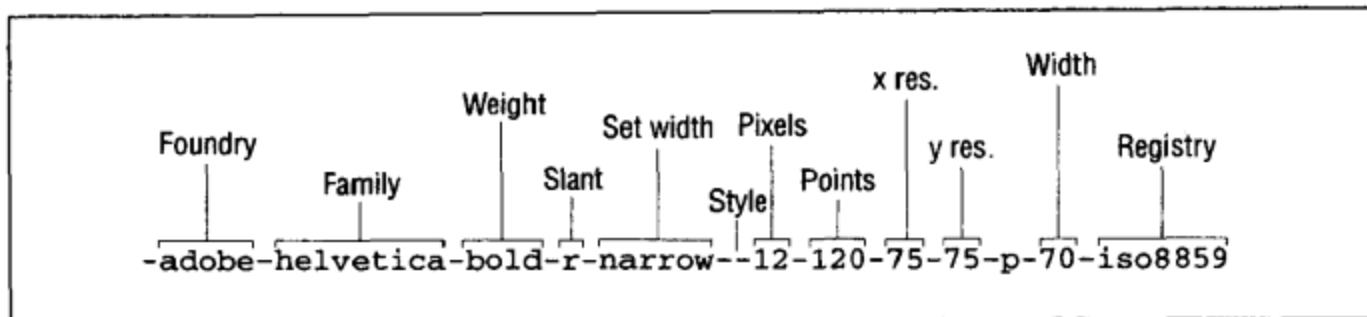


图 14-2 一个字体中的字段

幸运的是，我们不必记住大多数字段的用途。这些字段的任何一个都可以拥有通配符“\*”或“?”。但是确保连字符数量的正确很重要。在 X Window 系统中，有两个工具，一个是图形方式的 *fontsel*，另一个是名为 *xlsfonts* 的批命令，它们列出这些字段所有可用的组合，因此我们只需简单的挑选一个来用就行了。你主要只关注 foundry、family、weight、slant 和 points 等字段，而不必关心其余的字段。注意，“points”为点的十分之一，因此 120 就表示 12 点的字体。slant 可以是“l”表示斜体，或是“r”表示正常字体。你可以通过配置组件的字体属性来设置它的字体：

```
$label->configure (
    font => '-adobe-helvetica-medium-r-normal -80-75-75-p-46-*-*1');
```

一旦 Perl/Tk 移植到了 Windows 或 Mac 上，字体值就既可以以 XLFD 格式来指定，也可以使用简单的 Windows 风格来指定，如：Helvetica 24 bold。前一种格式将继续在所有的平台上得到支持。

## 图片

一些组件，如按钮和标签，可以显示双色位图或多色的像素图。由于同样的位图或图片可以用来装饰不止一个组件，Tk 把它们当作可以在一个或多个地点绘制的对象。也就是说图片对象包含数据，而组件却知道如何将它们在自己的空间中绘制出来。于是在一个组件上显示一幅位图或图片涉及两个步骤：给定图片文件来创建图片对象，和使用该图片对象配置组件的位图或像素图属性。

依据你所提供图片的类型，必须调用下面三种调用中的一种，来创建相应类型的图片对象：

```
# 只针对 XBM (X 位图)
$image = $label->Bitmap(file => 'face.xbm');

# 只针对 XPM (X 像素图)
$image = $label->Pixmap(file => 'smiley.xpm');

# 针对 GIF 或 PPM (可移植像素图) 格式, 使用照片构造器
$image = $label->Photo(file => 'frown.gif');
```

现在你就可以很容易的改变标签的图片了:

```
$label->configure (image => $image);
```

注意, 如果图片是一个位图, 那么你必须使用“bitmap”选项, 而如果是 XPM 或 GIF 文件, 那么就使用“image”属性。在是位图的情况下, 由选项“foreground”和“background”来表示两种颜色; 对于图片来说, 由文件提供它自己的颜色。

## 颜色

颜色可以用诸如“red”和“yellow”的符号名来指定。在 X 安装时的库目录中有一个名为 *rgb.txt* 的文件, 其中枚举了所有可用的符号名。另外, 你也可以用 #RGB、#RRGGBB、#RRRGGBBB 或 #RRRRGGGGBBBB 的形式指定 RGB 值, 这里每个 R、G 或 B 分别代表一个十六进制的表示红、绿、蓝程度的数字。

现在我们的题外话讲完了, 让我们简要的了解一下 Tk 和 Tix 的组件情况。

## 标签与按钮

表 A-1 中所描述的标准组件属性几乎覆盖了标签所提供的所有属性。这些属性的意思不言自明, 因此我们在这里也再不多说了。

按钮就是又多了一项属性——“command”选项的标签, 该属性可以把对按钮的点击与一个回调函数关联起来。下一个例子就描述了改变组件标签的回调过程 `change_label`:

```
use Tk;
$top = MainWindow->new();
```

```

$button = $stop->Button(text    => 'Start',
                        command => \&change_label);

$button->pack();
MainLoop();
sub change_label {
    $button->cget('text') eq "Start"      ?      # 创建
    $button->configure(text => 'Stop') :
    $button->configure(text => 'Start');
}

```

方法 `cget` 用来获取一个可配置属性的值。

这个回调函数也可以是一个闭包，如（我们省去了多余的代码）：

```

$button = $stop->Button(
    text    => 'Start',
    command => sub {
        $button->cget('text') eq "tart"      ?
        $button->configure(text => 'Stop') :
        $button->configure(text => 'Start')
    }
)

```

第三种配置 `command` 属性的方法，就是给它一个数组，第一个元素为回调过程，其他元素将在回调函数被调用时作为参数传递给它：

```

$button->configure (command => [\&change_label, "new label"]);

```

我们将在本章的后半部分由应用程序定义的卷滚中使用到这种方法。

## 单选按钮与复选按钮

单选按钮 (`radiobutton`) 是这样一种组件，它显示一个文本字符串、位图或图片，还有一个称做指示器 (`indicator`) 的菱形符号 (请看图 14-3)。它像按钮一样支持 “`command`” 属性。然而与按钮不同的是，它们一般成组进行使用，并可以让用户从众多选项中选择一个。因此单选按钮提供两个称做 `variable` 与 `value` 的属性，用来保持与同组中其他选项的同步，只有一个指示器处于选中的状态。如果你点击一个单选按钮，将会激活指示器并将与其相关的变量值改变为它自己的值

属性。与之相反，如果变量的值被改变了，单选按钮就会检查它是否与自己的值属性相匹配；如果匹配，它就会将自己的指示器打开。也许你已经猜到了，它在内部使用 tie 机制来监控对变量的改变。



图 14-3 单选按钮的例子

下面的例子创建了一个单选按钮组，\$bev 为同步变量。

```
$bev = "coffee";                                # 初始值
$coffee = $top->Radiobutton ( variable => \$bev,
                                text    => 'Coffee',
                                value   => 'Coffee');

$tea     = $top->Radiobutton ( variable => \$bev,
                                text    => 'Tea',
                                value   => 'tea');

$milk    = $top->Radiobutton ( variable => \$bev,
                                text    => 'Milk',
                                value   => 'milk');

# 摆放单选按钮
$coffee->pack (side => 'left');
$tea->pack    (side => 'left');
$milk->pack   (side => 'left');
```

因为单选按钮拥有不同的值，而且它们还共享相同的变量，我们可以确保在任何时候只有一个指示器被打开。

请参考表 A-3 来查阅更多的单选按钮的属性和方法。

复选按钮 (checkboxbutton) 与单选按钮非常相似。它有一个四方型的指示器，可根据关联变量值来进行切换。与单选按钮不同，切换它的值并不要求改变其他复选按钮的值，尽管你可以很容易地安排这么做。复选按钮适于应用在你想让用户选择所有合适选项的地方。



## Canvas

Canvas (画板) 组件实现了结构化的制图功能。它提供方法来创建和操纵各种图形元素 (*item*)，如圆、三角形、弧形、线、位图、多边形以及文本。它甚至允许你嵌入其他组件而且将它们当作普通的 canvas 元素来对待。

与 Java 对抽象窗口工具包 (AWT) 中 canvas 所提供的支持不同 (而且就我所知，几乎与其他所有的 GUI 工具包都不同)，Tk 的 canvas 元素本身就是对象：它们就像组件一样支持可配置属性，而且允许这些属性施加给单个或一组命名的元素。它们还可以与回调函数进行关联。(你可以编写这样一个有效的例子：“在鼠标划过这个圆时，调用过程 `foo`。”)

canvas 的元素又与组件不同，每个组件在 X 服务器上都有自己的窗口，而它们却没有。还有就是 canvas 元素并不参与布局管理 (它们无法由其包容器进行缩放)。我总是弄不明白这个工具包为什么要制造出这种差别来。例如，在 InterView 工具包中 (一个基于 C++ 的 X 窗口库，后来改名为 “Fresco”)，组件与结构化图形都继承自一个通用的称做 *glyph* 的图形对象。这让人感觉是一种更为一致的设计。当然，我还是要感谢这么良好的结构化图形实现能以免费的形式自由获得，而且还含有可让你立刻着手使用的优秀文档，相比之下，我的挑剔简直微不足道。

你可以调用 `Canvas::create` 方法在一个 canvas 组件中画一条线：

```
$top = MainWindow->new();
# 首先创建一个画板组件
$canvas = $top->Canvas(width => 200, height => 100)->pack();

# 在画板中画一条线
$id = $canvas->create ('line',
                      10, 10, 100, 100, # 从 x0,y0 到 x1, y1
                      fill => 'red'); # 对象填充色
```

`create` 命令的第一个参数为 canvas 元素的类型，而剩余的参数要依赖于这个元素。`create` 返回一个标识符，它可以在以后引用那个对象。比如，你可以使用 `coords` 方法来更新对象的坐标：

```
$canvas->coords ($id, 10, 100);
```



Tk 中的所有坐标都参照于左上角。x 坐标从左到右递增，而 y 坐标从上到下递增。

你可以使用 `move` 将对象移动到相对于当前位置的其他地方：

```
$canvas->move ($id, 15, 23); # 15 和 23 表示 x 与 y 偏移量
```

我们可以使用 `itemconfigure` 方法来配置 canvas 的元素；表 A-5 描述了任何一种元素的属性和方法，还描述了 canvas 作为一个整体时的属性和方法。

canvas 组件最为方便的功能之一，就是能够用一个字符串来标记一个或多个对象。一个对象可以使用任意个的标签来标记。标签字符串 `all` 代表所有的 canvas 对象。你可以在创建时标记对象，或使用 `addtag` 方法来进行标记。标签 `current` 代表鼠标正在它上面的那个元素。所有接受元素 ID 的 canvas 方法同样也接受一个字符串标签。例如要将所有标记为 “bunch” 的对象向右移动 10 个点，我们可以这么来写：

```
$canvas->move('bunch', 10, 0); # xoffset = 10, yoffset = 0
```

我们将在第十五章大量使用这种属性。

例 14-2 描述了一组中心沿阿基米德螺旋线画出的圆（如图 14-4）。阿基米德螺旋由方程  $r=a\theta$  来定义，这里  $r$  为半径，它（如图中的直线所指示的那样）与角  $\theta$  是成比例的。

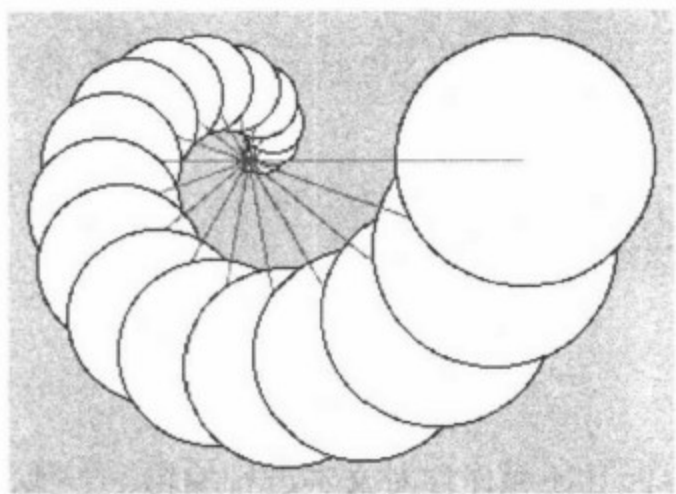


图 14-4 使用 canvas 绘制的结构化图形

为了增加视觉效果，我们也使圆的大小与角度成比例。

例 14-2: 画一个阿基米德螺旋

```
use Tk;
$top = MainWindow->new();
$canvas = $top->Canvas(width => 300, height => 245)->pack();
# 沿阿基米德螺旋绘制一系列的圆圈
# 这些圆圈的中心位于螺旋之上
# (radius of spiral = constant * theta)

$origin_x = 110; $origin_y = 70;           # 螺旋初始值
$PI = 3.1415926535;
$circle_radius = 5;                         # 第一个圆圈的半径
$path_radius = 0;
for ($angle = 0; $angle <= 180;
    $path_radius += 7, $circle_radius += 3, $angle += 10)
{
    # 轨迹坐标偏移量: r.cos(q) and r.sin(q)
    # sin() 与 cos() 的角度以弧度来表示 (degrees*p/90)
    $path_x = $origin_x + $path_radius * cos ($angle * $PI / 90);
    $path_y = $origin_y - $path_radius * sin ($angle * $PI / 90);
    # path_x 与 path_y 为新圆圈的中心坐标
    # Canvas::create 以自顶向左再从底向右进行创建
    $canvas->create ('oval',
        $path_x - $circle_radius,
        $path_y - $circle_radius,
        $path_x + $circle_radius,
        $path_y + $circle_radius,
        fill => 'yellow');
    $canvas->create ('line',
        $origin_x, $origin_y,
        $path_x, $path_y,
        fill => 'slategray');
}
MainLoop();
```

## 文本框与输入条

文本框 (text) 组件显示一行或多行文本，而且还允许你对文本进行编辑。(它默认的键盘绑定是 Emacs 类型的，因此，那些仍在使用 vi 编辑器的人就没办法了。) 这个组件功能强大，足以应付 Web 浏览器的显示，而且有几个项目已经很好地做

到了这一点。Perl/Tk 发行版中就包含了一个名为 tkweb 的 Web 浏览器，而 Python 的创造者 Guido van Rossum 已经用 Python 和 Tk 编制出了一个名为 Grail 的能够执行 Python 小应用程序的 Web 浏览器。

在这一节，我们将简单的了解一下文本框组件的功能，在第十六章“GUI 实例：Man 阅读器”中我们会进行更为深入学习，并基于它来建立一个应用程序。

### 在绝对位置插入文本

当你想以编程的方式在某个位置插入一段文本，或是选定一个范围的内容时，就需要指定一个或多个索引。索引就是像“2.5”这样的一个字符串，它表示第2行，第6列（行号起始于1而列号起始于0）。下面的代码创建了一个文本框组件并在那个位置插入一个字符串：

```
$t = $top->Text(width => 80, height => 10)->pack();  
$t->insert('2.5', 'Sample');
```

### 在逻辑位置插入文本

文本框组件支持标记 (*mark*) 的概念，标记是一个用户定义的名字，它代表文本框组件内一个单一的位置。位置是指两个字符之间的缝隙，而不是一个行或列对。这使得在一个标记处插入一个字符非常方便。一个标记位置就是一个逻辑实体；它不随插入或删除而改变。该组件支持内建的标记名，如 insert（插入光标所在位置），current（离鼠标箭头最近的字符），wordend（单词的末尾，也就是插入光标要放置的地方），end（插入光标行的末尾）等等。这些标记名可以用来代替前面所提到的行号或列号：

```
$t->insert("end", "Sample");
```

# 在末尾插入文本

请你查阅表 A-6 前面的文字来了解有关索引说明符的详细情况。下面一节的代码创建了一个文本框组件并使用不同的索引说明符在不同的位置插入字符串。

### 使用相对索引进行插入

索引 (index) 还可以是相对于基索引的相对索引，例如：

```
$t->insert('insert +5',
          'Sample');           # 在插入光标后 5 个字符的位置
$t->insert('insert linestart', 'Sample'); # 到达插入光标位置然后是那一行的
                                     # 起始位置
```

## 使用标签来改变成块文本的属性

组件支持标签 (tag) 或标签风格的概念, 它们是些由用户定义并表示一组文本属性的字符串 (字体, 颜色, 点刻风格等等)。考虑下面的例子:

```
$text->tagConfigure('foo',
                   foreground => 'yellow', background => 'red');
```

字符串 “foo” 可以对应于组件中的一个或多个毗邻的文本范围, 如:

```
$text->tagAdd('foo', '3.5', '3.7');
```

这将加重显示位于第 3 行, 从字符索引 5 到 7 的连续的文本区间。用于指定范围的索引同样可以是绝对的, 也可以是相对的标记位置。例如, 下面的代码段用于改变插入光标所在行的属性:

```
$text->tagAdd('foo', 'insert linestart', 'insert lineend');
```

多个标签可以作用于相互覆盖的文本区间; 与之相反, 一个标签也可以作用于多个文本区间。所有的文本框组件都支持一种名为 sel 的特殊标签, 它表示当前选中的文本区间。你可以通过提供标签名作为 insert 的第三个参数, 来插入新的文本或进行文本的格式化:

```
$t->insert('3.5', 'Sample', 'foo');
```

文本框组件允许你嵌入任何其他的组件, 并把它们当作单一的字符来看待, 也就是说, 你在插入更多文本的同时, 可以在文本中嵌入按钮和列表框等组件, 并且可以使其同文本一起移动。

## 输入条组件

Perl/Tk 提供了一个名为输入条 (Entry) 的组件来完成单行的文本输入, 它并不支持标签、标记或嵌入窗口等内容。它本质上是一种轻量级版本的文本框组件。

Perl/Tk（而不是原始的Tk）还支持一种叫做TextUndo的组件，它是文本框组件的基类。它提供了无限次的回退机制（哎，没有“重做”机制；你无法回退一个回退操作！），并且提供了从文件中加载和向文件中保存的方法。这个组件并不是原始Tcl/Tk发行版中所包含的组件。

## 文本框组件与绑定

文本框组件支持TIEHANDLE和print方法，可以允许你将其用作一个模块来仿真文件句柄。下面就是一个例子，说明了如何使用这种机制将文件句柄的活动重定向到文本框组件：

```
use Tk;
my $mw = MainWindow->new;           # 创建一个顶层窗口
my $t = $mw->Scrolled('Text');      # 创建一个滚动文本窗口
$t->pack(-expand => 1,               # 配置它
        -fill => 'both');
tie (*TEXT, 'Tk::Text', $t);        # 将文件句柄与组件进行绑定
print TEXT 'Hi there\n';            # 这段文字将会出现在组件上
```

## 列表框

列表框（listbox）用于显示一组字符串，每行显示一个，如图14-5所示。所有的字符串都拥有相同的显示特征。如果你想混杂使用不同的字体和颜色，那么可以编写一个文本框组件的包裹层，来仿真一个特别的列表框组件。

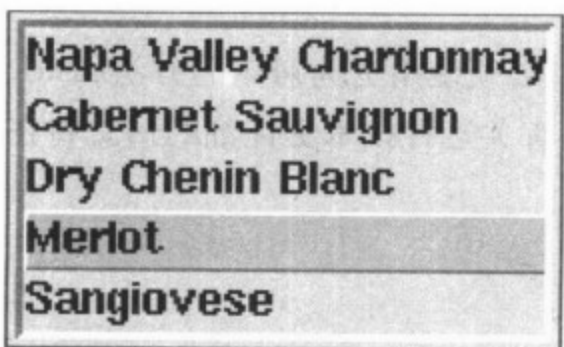


图 14-5 用于选择酒类的列表框

列表框默认的绑定功能为选择元素与取消选择。但是如果你需要其他的功能，比如鼠标双击时进行特殊的处理，这就要由你来负责将其绑定到你所选定的函数上

了。选择方式包括“单选”，“浏览方式（默认）”，“多选”和“扩展方式”。单选和浏览方式表示任何时候只能选择一个元素。浏览方式还允许你使用按钮1来拖曳选择。在多选模式下，你可以选择任意数量的元素；选择或取消选择一个元素，对其他的选中元素没有任何影响。扩展方式允许你通过点击并拖动的方式来选择多个元素，但是一次单个的点击操作，将会取消上一次的选并选择当前元素。

类似于文本框组件，除了对索引之外，列表框同样还有多种方式来标识列表的位置；比如有 `end` 和 `active`（代表当前光标所在的位置）等。表 A-8 描述了这些索引以及列表框的属性和方法等内容。例 14-3 创建了图 14-5 所描述的列表框。

例 14-3: 带有回调的列表框

```
use Tk;
$top = MainWindow->new();
$wine_list = $top->Listbox("width" => 20, "height" => 5
                           )->pack();
$wine_list->insert('end', # 将下面的列表插入到尾部
                  "Napa Valley Chardonnay", "Cabernet Sauvignon",
                  "Dry Chenin Blanc", "Merlot", "Sangiovese");
$wine_list->bind('<Double-1>', \&buy_wine);
sub buy_wine {
    my $wine = $wine_list->get('active');
    return if (!$wine); # 如果没有选中的列表元素就返回
    print "Ah, '$wine'. An excellent choice\n";
    # 从清单中将这种酒删除
    $wine_list->delete('active');
}
MainLoop();
```

列表框并不提供诸如 `command` 的属性，因此我们必须使用更为通用的方法 `bind`，来建立鼠标双击与用户定义子例程之间的关联。有关这种技术更详细的信息将在“事件联编”一节讲述。

## 图文框

图文框 (frame) 组件是一种相当乏味的组件，但是当你需要复杂的组件布局或创建复合组件时，它非常有用。

如果你有一个复杂的GUI窗体，你最好将屏幕分成几大块，每一块对应特定的功能，并将每一块放到它自己的图文框中。这样一来，你就可以在顶层方便的安排或重新安排这些部分。我们将在“布局管理”一节更多的谈到这个问题。图文框组件是一种包容器，可以用来创建其他的基础组件如按钮、文本框和滚动条等。

## 菜单

“菜单(menu)”这个字眼通常指这样一种安排，用户通过它可以点击一个菜单按钮，然后就会弹出一组标签或按钮组件。菜单共有三种类型：下拉菜单，选项菜单或弹出式菜单。

Tk提供了一种菜单按钮组件，当它被点击时可以弹出一个菜单组件。菜单组件就是菜单项组件的包容器；它并不代表整个菜单。我们将使用不同的字体来区分菜单与菜单项的概念。图 14-6 描述了这些部件。

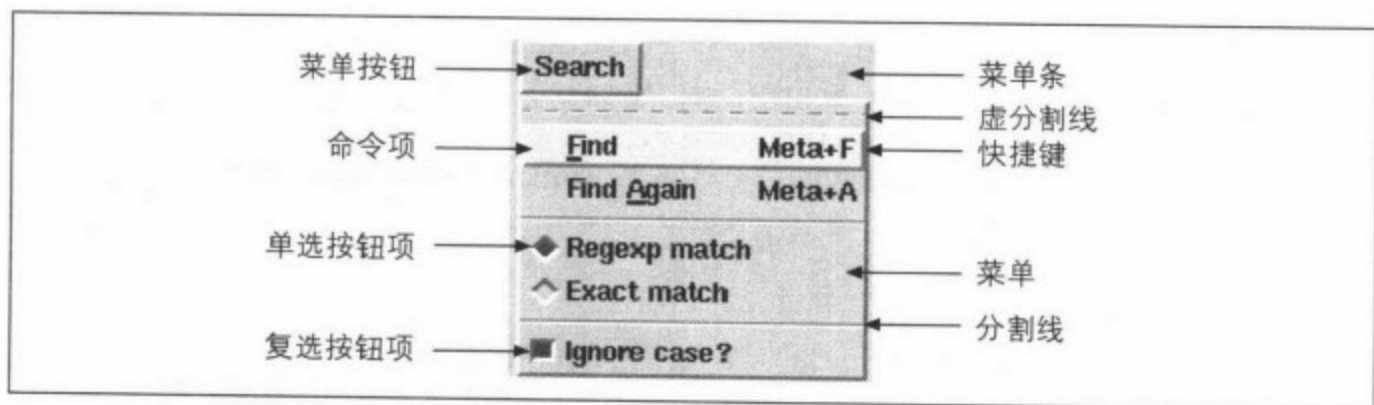


图 14-6 下拉菜单和菜单按钮

要构造一个菜单，你需要完成下列步骤：

1. 创建一个菜单条来包容菜单按钮组件。菜单条就是一种普通的图文框组件。
2. 创建一个或多个菜单按钮并把它们安放在菜单条中。
3. 让菜单组件来创建和管理菜单输入组件。

菜单按钮组件和菜单组件的属性与 API 分别在表 A-9 和表 A-10 中列出。例 14-6 描述了图 14-6 中的菜单是如何创建的。



例 14-4: 用于进行文本搜索的下拉菜单

```

use Tk;
$stop = MainWindow->new();
# 使用一个图文框来作为菜单按钮的容器
$menu_bar = $stop->Frame()->pack(side => 'top');

# "Search" 菜单按钮
$search_mb = $menu_bar->Menubutton(text      => 'Search',
                                   relief      => 'raised',
                                   borderwidth => 2,
                                   )->pack(side => 'left',
                                           padx => 2
                                           );

# "Find" 菜单按钮
$search_mb->command(label      => 'Find',
                    accelerator => 'Meta+F',
                    underline   => 0,
                    command     => sub {print "find\n"}
                    );

# "Find Again" 菜单按钮
$search_mb->command(label      => 'Find Again',
                    accelerator => 'Meta+A',
                    underline   => 5,
                    command     => sub {print "find again\n"}
                    );

$search_mb->separator();
$match_type = 'regexp';      # 默认的查找模式为 regexp
$case_type = 1;              # 默认情况下忽略大小写 (激活点选按钮)
# Regexp match
$search_mb->radiobutton(label    => 'Regexp match',
                        value     => 'regexp',
                        variable  => \ $match_type);

# Exact match
$search_mb->radiobutton(label    => 'Exact match',
                        value     => 'exact',
                        variable  => \ $match_type);

$search_mb->separator();
# Ignore case
$search_mb->checkbox(label    => 'Ignore case?',
                    variable => \ $case_type);

MainLoop();

```



这个例子调用了菜单按钮组件 (\$search\_mb) 的多个方法，如：command、separator、checkbutton 和 cascade 等。实在是奇怪，这些方法实际上属于菜单组件的接口，而不是菜单按钮的（请查看表 A-9 和表 A-10）。为了方便，Perl/Tk 的菜单按钮接受这些命令，并悄悄的将其分派给它所关联的菜单组件来完成。

通常情况下，菜单项按照它们创建的顺序进行摆放，但是你可以使用 add 方法来显式的指定索引位置。其索引语法与列表框组件相似并在附录一中有描述。我们将在第十六章使用这种方法动态的创建菜单。

## 滚动条和滚动

尽管滚动条 (scrollbar) 是全功能的组件，但是它们很少独立使用；它们通常被用来控制相关的组件。由于这种紧密的关联性，Perl/Tk 提供了一个方便的函数称做 Scrolled，将滚动条绑定到你所选择的组件上，而不必再显式的创建，变换尺寸和安放它们了。下面的例子创建了一个支持滚动的列表框：

```
$scrolled_list = $top->Scrolled('Listbox', listbox options,  
                                scrollbar => 'se');
```

在内部，这段代码创建了一个图文框组件，水平和垂直滚动条（如果需要的话）以及一个列表框并将它们摆放在一起，然后返回指向该图文框组件（容器）的引用。这样是不是很棒？实际上，对于大多数情况下的列表框和滚动文本框，Perl/Tk 分别提供了方便的称做 ScrListBox 和 ScrLText 的方法，从而更进一步减少你输入的烦琐：

```
$scrolled_list = $top->ScrListBox(listbox options);
```

你通常所要了解滚动的知识就这么多，你可以直接跳到“标尺”一节而不会损失连贯性。

## 定制的滚动

存在这样的情形，那就是需要自己来处理滚动。例如，你有三个列表框并且想进行同步的滚动。这就意味着，每当滑块移动时你需要安排让滚动条发送消息给所

有这三个组件。这里的微妙之处就是反过来也是可行的：这些组件在它们自己通过其他方式而发生滚动时，也要发送消息。例如，如果你点击一个列表框并拖曳光标的话，列表框就会滚动它所包含的内容。那么它就必须确保滚动条与另外的两个列表框的同步。换言之就是滚动条并不总是控制者；它非常类似于一种“我可以滚动你，你也可以滚动我”的关系。

如表A-11所示，没有一个显式的用于将滚动条绑定到一个组件上的属性，但是滚动条确实有一个名为command的回调属性，它可以在滑块移动时得到通知。同时，巧的是所有可以滚动的组件（列表框，文本框组件，图文框以及画板）都支持两个名为xview和yview（表A-12）的方法，由它来告诉可滚动组件它的哪一部分内容显示在窗口中。因此，要想让一个滚动条发送消息给一个使自己滚动的组件的话，我们如下来配置滚动条的command属性：

```
$scrollbar->configure (command => [N$widget]);
```

滚动条会自动地调用指定的组件上的方法（xview或yview）。那么组件是如何知道要向哪里滚动呢？噢，你不知道吧，是滚动条给yview调用提供了一些参数，这样在内部从滚动条发送给组件的消息就大概是下面的这个样子：

```
$widget->yview('moveto', 30);
```

它告诉这个组件将自己的内容进行调整，使顶部的行或像素处于30%的位置。

现在让我们再来看一下组件通知滚动条的问题。

所有可以滚动的组件都支持两个名为xscrollcommand和yscrollcommand的方法，它们应配置来调用滚动条的set方法，如下所示：

```
$listbox->configure ('yscrollcommand', [$scrollbar]);
```

图14-7描述了这种共生的关系。表A-11和A-12中提供了对这些命令及属性的详细描述。

注意，在这个例子中，你不必使每个列表框都来驱动另外的两个列表框。只要它们都驱动滚动条就行了，因为滚动条与另外两个是绑定在一起的。

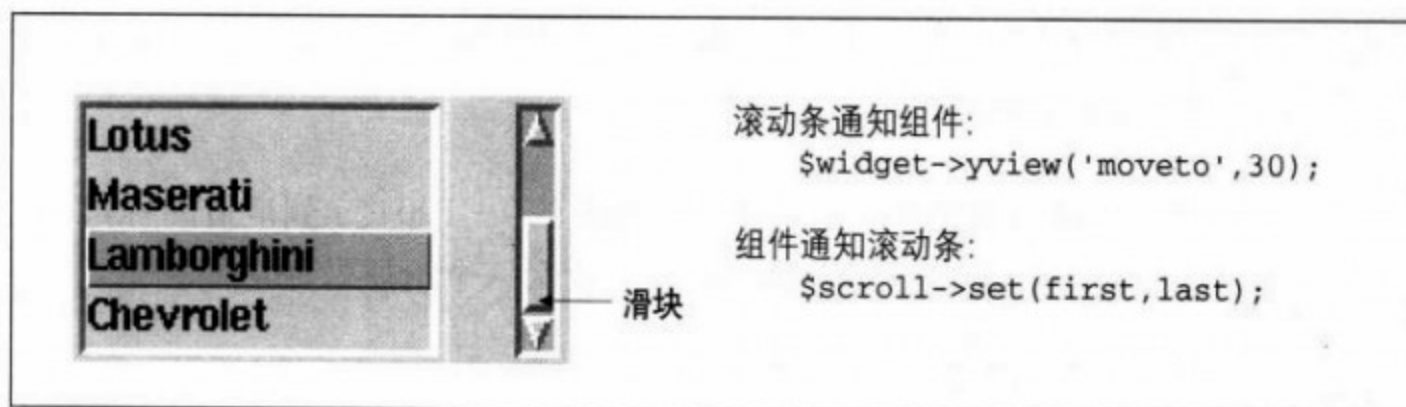


图 14-7 一个滚动条与其关联组件（列表框）之间的交互

例 14-5 针对一个列表将各种配置命令放在一起使用。

例 14-5: 设置一个滚动条和列表框使它们可以相互滚动

```
use Tk;
$top = MainWindow->new();
$car_list = $top->Listbox("width" => 15, "height" => 4,
                        )->pack(side => 'left',
                                padx => 10);

$car_list->insert('end', # 在末尾插入下面的列表
                "Acura", "BMW", "Ferrari", "Lotus", "Maserati",
                "Lamborghini", "Chevrolet"
                );

# 创建一个滚动条，并告诉它列表框的存在
$scroll = $top->Scrollbar(orient => 'vertical',
                        width => 10,
                        command => ['yview', $car_list]
                        )->pack(side => 'left',
                                fill => 'y',
                                padx => 10);

# 告诉列表框滚动条的存在
$car_list->configure(yscrollcommand => ['set', $scroll]);
MainLoop();
```

## 标尺

标尺 (scale) 组件就像一个温度计。它沿水平或垂直的槽边显示刻度并在槽的中

间提供一个滑块 (slider)，可以通过编程或手工（使用鼠标或键盘）的方式进行滑动。表 A-13 描述了标尺的属性和方法。

图 14-8 展示了两个显示摄氏和华氏（对应于摄氏的 0 ~ 100 度）刻度值的标尺。这两个标尺相互协调，这样一个滑块的移动将导致另一个滑块做相应的移动。

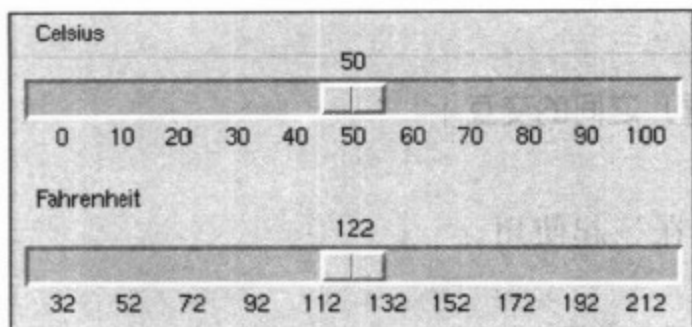


图 14-8 协调一致的摄氏与华氏标尺

例 14-6 描述了它们的一种实现

例 14-6: 使用两个标尺来完成摄氏与华氏的转换

```
use Tk;
# 使用标尺来展示摄氏与华氏之间的对应关系
$stop = MainWindow->new();

$celsius_val = 50;
compute_fahrenheit();
#-----CELSIUS Scale-----
$stop->Scale(orient      => 'horizontal',
             from        => 0,                # 从摄氏 0 度
             to          => 100,              # 到摄氏 100 度
             tickinterval => 10,
             label       => 'Celsius',
             font        => '-adobe-helvetica-medium-r-normal' .
                           . '--10-100-75-75-p-56-iso8859-1',
             length      => 300,              # 像素数
             variable    => \$celsius_val,    # 全局变量
             command     => \&compute_fahrenheit # 改变华氏值
             )->pack(side => 'top',
                    fill => 'x');
#-----华氏标尺-----
$stop->Scale(orient      => 'horizontal',
             from        => 32,                # 从华氏 32 度
```

```
to          => 212,          # 到华氏 212 度
tickinterval => 20,          # 每 20 度一个间隔
label       => 'Fahrenheit',
font        => '-adobe-helvetica-medium-r-normal'
            . '--10-100-75-75-p-56-iso8859-1',
length      => 300,          # In pixels
variable    => \ $fahrenheit_val, # 全局变量
command     => \&compute_celsius # 改变摄氏值
)->pack(side => 'top',
        fill => 'x',
        pady => '5');

sub compute_celsius {
    # 摄氏标尺的滑块在
    # $celsius_val 被改变时将会自动移动
    $celsius_val = ($fahrenheit_val - 32)*5/9;
}

sub compute_fahrenheit {
    $fahrenheit_val = ($celsius_val * 9 / 5) + 32;
}

MainLoop();
```

在这个例子中，摄氏标尺在它的滑块移动时调用 `compute_fahrenheit()`。这个过程将会改变 `$fahrenheit_val` 的值，而它又与华氏标尺相关联。如你所见，一般来说使用 `command` 和 `variable` 属性就足够让标尺工作起来。你不必显式的调用 `set()` 方法。

## 树形列表组件

层次结构的数据，如文件系统结构或组织图等，可以使用树形列表组件 `HList` 来表示。每个条目 (entry) 都相对于父条目缩进一个层次。树形列表组件还可以随意的画出分支，并可以将图标或其他组件与每个条目关联起来。一个条目不是以索引 (如列表框组件就是这样)，而是以其“条目路径”进行标识的，路径条目就像文件路径一样带有你所选择的分隔符。表 A-14 描述了有关这个组件的一些有趣的属性和方法。

例 14-7 使用树形列表组件构建了一个目录浏览器。双击一个目录条目将会展开或折叠目录并对图标做相应的改变。

例 14-7 描述了一种构建图 14-9 所示应用的方法。请你特别注意打开及设置位图的代码，以及在执行程序时用于改变光标形状的那一部分程序。

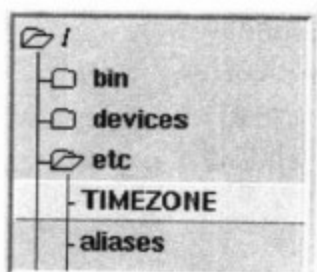


图 14-9 应用在一个目录浏览中的树形列表组件

例 14-7: 使用树形列表组件来构建目录浏览器

```
use Tk;
require Tk::HList;
$stop = MainWindow->new();
$shlist = $stop->Scrolled('HList',
                           drawbranch    => 1,      # 是的，画出分支
                           separator     => '/',     # 文件名分隔符
                           indent        => 15,      # 像素
                           command       => \&show_or_hide_dir);
$shlist->pack(fill => 'both', expand => 'y');
# 读取位图文件并创建 "image" 对象
$open_folder_bitmap = $stop->Bitmap(file => './open_folder.xbm');
$closed_folder_bitmap = $stop->Bitmap(file => './folder.xbm');

# 从根目录开始
show_or_hide_dir("/");
MainLoop();
#-----
sub show_or_hide_dir {          # 在条目被双击时调用它
    my $path = $_[0];
    return if (! -d $path);    # 不是一个目录
    if ($shlist->info('exists', $path)) {
        # 切换目录的状态
        # 如果下一个条目为当前路径的子串，那么
        # 我们就知道这个目录是打开的
        $next_entry = $shlist->info('next', $path);
        if (!$next_entry || (index ($next_entry, "$path/") == -1)) {
            # 否，将其打开
            $shlist->entryconfigure($path, image => $open_folder_bitmap);
            add_dir_contents($path);
        } else {
```

```

        # 是的, 通过改变图表来将其关闭并删除其子节点
        $hlist->entryconfigure($path,
                                image => $closed_folder_bitmap);
        $hlist->delete('offsprings', $path);
    }
} else {
    die "'$path' is not a directory\n" if (! -d $path);
    $hlist->add($path, itemtype => 'imagetext',
                image      => $icons{"open"},
                text       => $path );
    add_dir_contents($path);
}
}

sub add_dir_contents {
    my $path = $_[0];
    my $oldcursor = $stop->cget('cursor'); # 记住当前的光标并
    $stop->configure(cursor => 'watch');   # 将其改为 watch 光标
    $stop->update();
    my @files = glob "$path/*";
    foreach $file (@files) {
        $file =~ s|//|/|g;
        ($text = $file) =~ s|^\.*/||g;
        if (-d $file) {
            $hlist->add($file, itemtype => 'imagetext',
                        image => $icons{"closed"}, text => $text);
        } else {
            $hlist->add($file, itemtype => 'text',
                        text => $text);
        }
    }
    $stop->configure(cursor => $oldcursor);
}

```

我们现在就结束了有关 Tk 和 Tix 组件的旅行。请查阅 Tk 文档来了解其他的组件，查询 Tk 发行版中的 *contrib* 目录下了解贡献的组件。下面让我们来了解一下 Tk 所提供的其他机制：布局管理、定时器、事件联编和事件循环。

## 布局管理

你已经看到了 pack 方法的用途。我们所要讲述的就是“布局管理”，这是一种在



屏幕上安排组件，并在屏幕尺寸改变时指定策略以重新安排这些组件的艺术。Tk 支持三种类型的布局管理器：放置器 (placer)，打包器 (packer) 和栅格 (grid)。放置器是最简单的一种。与 Motif 中的广告牌组件，或 Visual Basic 的布局管理策略一样，你必须指定每个组件的  $x$  和  $y$  坐标。我希望你参考 Tk 的文档来了解有关放置器更详细的信息。

## 打包器

打包器，就像 Motif 的窗体组件，是一种功能强大的基于约束的布局管理器。打包器不是一种对象。它只是一种由 `pack()` 方法实现的算法。换句话说，对 `$widget->pack()` 的调用就是一个发送给组件的请求，将它自己安放在其包容组件中下一个可用空间中。

当你打包一个手提箱时，你通常是从一端开始将每一个物品紧挨着排满剩余的空间。打包器的工作与此非常类似，但有一个关键的不同点。一旦你将一个组件排放到包容器的一边，它就会贴着两边占据整个半块空间。图 14-10 描述了这种打包算法。

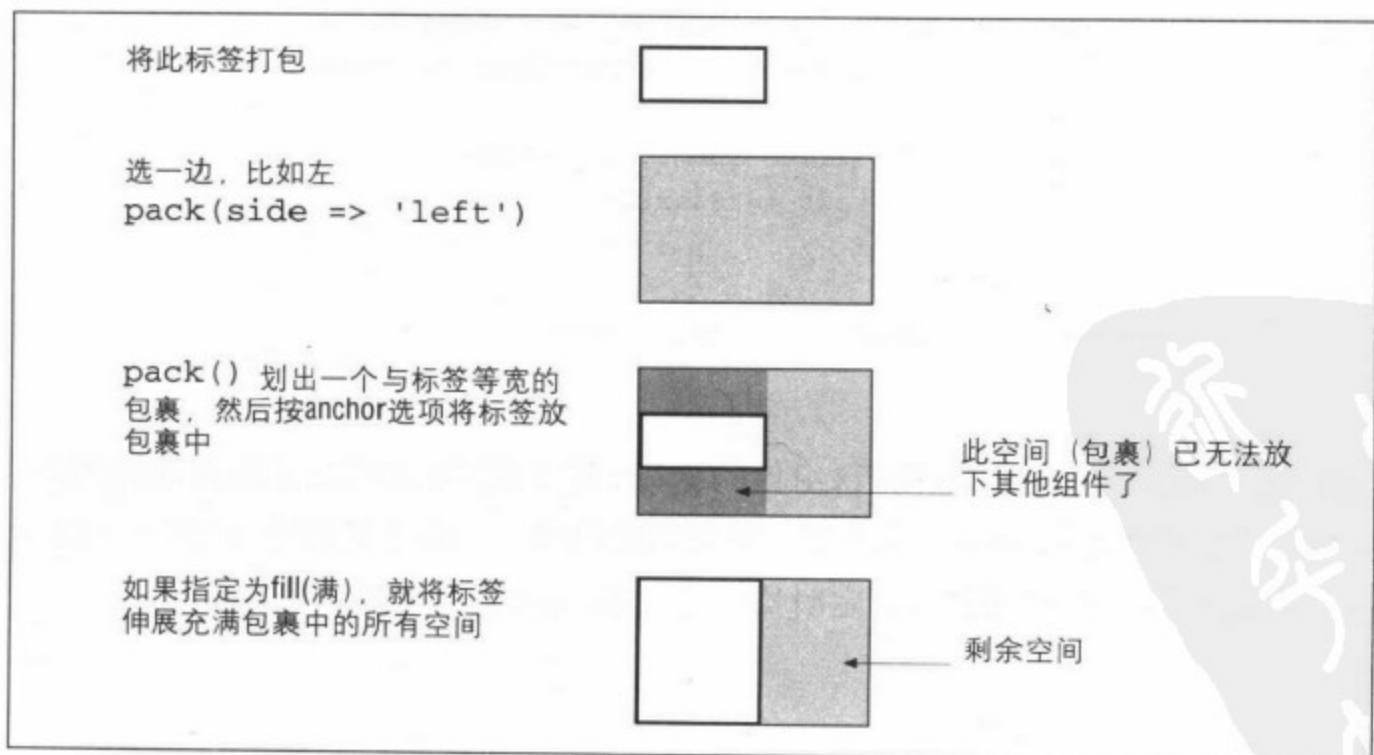


图 14-10 打包算法



在此图中，如果指定的边为顶部或底部，那么标签的高度将表示所占据包裹的高度。

你可以使用 `pack` 来完成三件事情：

- 指定组件将要打包的顺序。你调用 `pack` 的顺序决定了打包的顺序，从而也就决定了一个组件可以存取的空间大小。如果包容器调整了大小，打包算法将会按照相同的次序重新执行。
- 指定如何填充包裹空间。这个选项由填充值指示：`x`（在 `x` 方向扩展组件且宽度充满包裹空间），`y`（扩展并占满高度空间），`both` 或 `none`。`ipadx` 和 `ipady` 选项用于保留围绕组件的一些内部填充空间，因此分配的包裹空间可以比组件尺寸本身所需要的空间大得多。`anchor` 选项指定了组件在包裹空间中所停靠的边缘或角落，默认为 “center”。
- 指定在所有的组件都被插入后，对留给包容器（`parent`）剩余空间的处理方式。它由 `expand` 参数来提供。通常，最后一个插入的组件将得到所有剩余的空间并且填满那块空间。但是如果其他的组件打包时指定了扩展值 “`y`”（是），那么任何剩余的水平空间，都将被所有指定该选项且 `side` 值为 `left` 或 `right` 的组件平均分配。与之相似，多余的垂直空间将在所有拥有值 “`top`” 或 “`bottom`” 的组件间平分。注意，打包算法决不会使组件相互重叠。

如果第一个插入的组件将占据整个一边空间，你也许会纳闷如何该在左边插入三个组件。

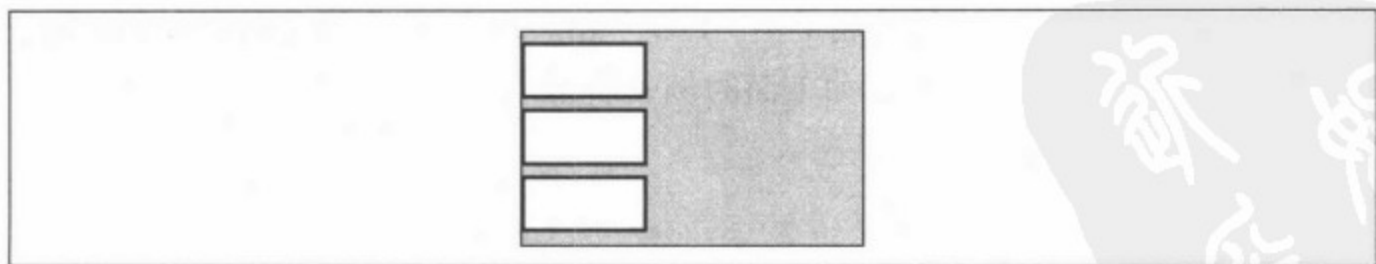


图 14-11 在左边安放三个组件

唯一解决这个问题的办法就是创建一个图文框组件并让它停靠在顶层窗口的左半边。既然图文框就是其他组件的包容器，因此这三个组件就可以打包在这个图文框中，实现代码如下：

```
$frame->pack(-side => 'left', -fill => 'y', -expand => 'y');  
# 现在创建三个按钮 b1, b2 和 b3 作为图文框的子组件  
# 并从上往下进行排放  
$b1 = $frame->Button (-text => 'Oh    ')->pack();  
$b2 = $frame->Button (-text => 'Hello')->pack();  
$b3 = $frame->Button (-text => 'There')->pack();
```

pack 默认从顶部往下插入组件。

作为另一种解决方案，你也许会觉得使用栅格布局管理器更容易些。

## 栅格

grid 方法使组件使用栅格布局管理器。位于同一个容器中的所有组件，必须使用相同的布局管理器，但是你却有自由为给定的双亲组件及其所包容的组件的组合，或嵌套在子组件中的组件，选择布局管理器。

栅格布局管理器允许你将组件以行列的形式进行存放，非常类似于HTML的table 标签。一列中任意组件的最大宽度决定了该列的宽度，而行的高度有那一行中高度最大的组件决定。下面就是你如何来使用栅格布局管理器的：

```
$button->grid (row => 0, column => 0);
```

这条命令将按钮安放在左上角。

与HTML的表相似，通过使用 rowspan 和 colspan 选项，一个组件可以跨越任意数量的行或列。组件仍然属于由“row”和“column”指定的行和列，但是却占据所要求的行数和列数，如下面的例子所示：

```
$button->grid(row => 1, col => 2,  
             colspan => 2, sticky => 'ns');
```

这个例子中的按钮占据了两列但还没有用完所有的空间。本例中，sticky选项用来指示栅格使组件紧挨单元的北面 and 南面存放。如果你将它配置为“nsew”，那么组件将扩展以填充整个单元空间。默认情况下，组件居中排放在包裹空间中并只占据所需要的空间。

与打包器类似，栅格同样理解 `ipadx` 和 `ipady` 选项。

## 定时器

Tk 支持一种轻量级的定时机制，可以由它在规定的延时（单位为毫秒）后回调一个过程函数。你可以使用 `after` 方法来创建一次性定时器；而使用 `repeat` 来创建重复性的定时器。在下面的例子中，按钮将在被按下时改变其标签，并在 300 毫秒后恢复：

```
$button->configure (text => 'hello',
                    command => \&change_title));
sub change_title {
    my ($old_title) = $button->cget('text');
    $button->configure (text => 'Ouch');
    $button->after (300,
                  sub {$button->configure(text => $old_title)});
}
```

我们将在第十五章的动画中大量的使用 `after`。

`after` 和 `repeat` 都返回定时器对象。由于这些机制相当的高效和轻量（与使用 `alarm()` 和 `SIGALRM` 不同），你可以拥有大量的定时器。要取消一个定时器，使用方法 `cancel`：

```
$timer = $button->after(100, sub {print "foo"});
$timer->cancel();
```

Tk 的超时机制与 `SIGALRM` 不同，并不是抢先的。它需要控制返回事件循环之后，才能检查下一个定时器是否到点了。一个运行时间长的子例程将会延迟定时事件。

## 事件联编

事件联编将一个回调函数与任意类型的事件关联起来。你已经见到了事件联编的实例——如按钮组件的 `command` 属性，安排了一个用户定义的在鼠标点击时调用的过程程序。`bind()` 方法提供了一种更为通用的存取大多数基础事件的方

法，如键盘和鼠标按键按下等事件。（一次鼠标的点击是一次按下并释放的操作，因此我们这里谈论的事件相当的底层。）其他“有趣”的事件类型包括鼠标移动，鼠标指针进入或离开一个窗口，以及窗口在显示器上的显现或调整大小等。所有的组件自身都依赖于bind方法来实现它们自己的功能，而且还能让你自己创建额外的联编。如果你所跟踪的事件，在组件中发生或与组件有关（如调整窗口大小），那个绑定的过程将被执行。

bind 的语法如下所示：

```
$widget->bind(event sequence, callback);
```

事件序列就是包含一个基本事件序列的字符串，其中的每个基本事件都包含在尖括号中。下面是一些事件序列的例子：

```
"<a>"           # 键 "a" 被按下 (Control/shift/meta 没有
                  # 被按下)
"<Control-a>"     # Control 与 a 一起被按下
"<Escape> <Control-a>" # 双事件序列
"<Button1>"       # 鼠标按钮 1 被点击
"<Button1-Motion>" # 在鼠标按钮 1 被按下时移动鼠标
```

一个单一的事件（包含在尖括号中的）具有下列的通用语法：

```
"<modifier-modifier...-modifier-type-detail>"
```

修饰符的例子有Control、Meta、Alt、Shift、Button1（或B1）、Button2、Double（双击）和Triple等。修饰符Any是一个通配符——所有可能的修饰符（包括没有修饰符的情况）都匹配基本事件。

事件类型是下列类型的一种：KeyPress、KeyRelease、ButtonPress（或Button）、ButtonRelease、Enter、Leave和Motion。

对于键盘事件说明来说，其中细节为一个描述确切按键的文本字符串。X窗口系统称之为*keysym*。对于可以打印出来的ASCII字符，*keysym*就是打印字符本身。其他*keysym*的例子有Enter、Right、Pickup、Delete、BackSpace、Escape、Help、F1（功能键）等等。

最常用的事件类型就是按键和鼠标点击，因此Tk允许一种简写形式的联编：你可以使用 `<a>` 来代替 `<KeyPress-a>`、`<l>` 来代替 `<Button1-ButtonPress>`。

文本框与画板组件支持更为精细的联编。它们不仅支持对组件本身的联编还支持对不同标签的事件联编。`bind` 允许你指定标签名为第一个参数，事件序列与回调函数分别为第二，三个参数：

```
$text->bind('hyper-link', '<l>', \&open_page);
```

这段代码确保任何标记为“hyper-link”的文本，均响应点击事件并调用过程 `open_page`。

## 多重联编

有可能让几个联编共同响应同一个事件。例如，当按下鼠标按键时，`<Button1>` 与 `<Double-Button1>` 均可作为后选。如果对于一个给定的组件（或标签）存在冲突，原则上最为特殊的联编将被执行。`<Double-Button1>` 要比 `<Button1>` 更为特殊，因为它的说明更长一些。

除了在组件层次上匹配最为特殊的联编，Tk还将在类层次上匹配最为特殊的联编（比如，代表所有按钮的类），然后是在顶层的组件层次上，最后是在称做“all”的层次上进行匹配。所有四类联编均被执行。顺序本身可以使用 `bindtags()` 方法进行改变，但是我建议你不要做这种事情。

尽管Tk允许你改变默认的组件联编，我还是建议你不要去改动它们，因为人们已经习惯了某种惯常的方式。例如，在文本框组件中双击鼠标，通常会选中鼠标指针下面的单词，如果你碰巧改变了这种行为，用户就会觉得特别不舒服。另一方面，有许多其他的地方，你可以并且需要加入你自己的联编。正如我们将要在下面两章看到的，画板和文本框组件标签是改动最为频繁的事件联编对象。

## 事件的细节

我们已经见到了如何精确的指定一个事件。有时我们需要做完全相反的事情——使事件限定能最大限度的通用，比如 `<Any-KeyPress>`。例如，你大概不想为键

盘上的每一个字符都指定一个唯一的联编吧。但当一个键被按下时，回调函数或许想知道是哪个键被按下了。这里就需要事件的细节。

每个事件都包含有与那个事件相关的所有细节，而函数 `Ev()` 被用来获取所有的这些细节。`Ev()` 的参数是一个单一的字符，由它来指定你对事件记录的哪一部分感兴趣。`Ev('K')` 指定键盘编码，`Ev('x')` 和 `Ev('y')` 指定鼠标指针的坐标，而 `Ev('t')` 指定事件发生的时间。`Ev` 的这种参数约有 30 种之多。下面的代码演示了你如何来使用这种机制：

```
$label->bind("<Any-KeyPress>" => [\&move, Ev('k')]);
sub move {
    my $key = pop;
    if ($key eq 'k') {
        move_left();
    } elsif ($key eq 'l') {
        move_right();
    }
}
```

在这个例子中，`bind` 说明注册了它对键盘事件的兴趣，并且指定了调用回调函数时要提供键盘编码。

## 事件循环

`MainLoop` 执行一个事件循环，由它从下面的窗口系统中采集事件，并分发给相应的组件。当为了响应一个事件而调用一个回调函数时，尽快返回（或“释放”）是回调函数的责任。否则它将会阻止所有从那一刻开始到来的事件。

对于 CPU 消耗比较厉害的长时间运行的活动来说，你有责任将其切分成可以管理的小段，并安排一个定时器每隔一段时间调用一次处理例程。这样可以给事件循环一个分发其他未决事件的机会。这种类型的 CPU 共享被称做协同多任务。微软早期的 Windows（直到版本 3.1）与之类似，要依赖于你的应用要是一个好公民；否则它就会暂时中止整个操作系统。

对于射线跟踪和动画这类既需要大量的 CPU，又需要大量的 GUI 操作的任务来说，你可以使用 `$widget->update` 方法来处理所有的事件。该方法直到等待在事件队列中的所有事件处理完时才会返回。

我们以前在第十二章就讨论过，阻塞的系统调用在事件驱动的环境中可不是什么好东西。其中最为常见的就是与管道和套接字交流的 `read` 和 `write`。例如，菱形操作符只有在它获得一行文本时才会返回。你必须让 Perl/Tk 来告诉你何时调用不阻塞，而不是直接调用一个 I/O 调用。Tk 提供了一个名为 `fileevent` 的过程，它能够在文件描述符可读或可写时通知回调函数。下面是使用它的一个例子：

```
open (F, "/tmp/foo");
$button->fileevent(F, "readable", \&read_file);
sub read_file {
    if (eof(F)) {
        $button->fileevent(F, "readable", undef); # 取消联编
        return ;
    }
    if (sysread (F, $buf, 1024)) {
        $text->insert('end', $buf); # 追加读取的数据
    } else {
        # sysread 返回 undef。文件有问题
        $text->insert('end', "ERROR !!!");
        $button->fileevent(F, "readable", undef); # 取消联编
    }
}
```

当回调函数被调用时，Tk（在 Unix 系统中内部使用的是 `select`）将确保至多有一个字符可供读出或写入。除此之外，它可能阻塞也可能不阻塞，不做任何保证。当出现文件末尾或错误时，回调函数也会被调用，因此你必须此加以检查。否则，当回调返回时将会被再次调用，这将导致无限循环。正如我们在讲述网络计算的几章中所讲的那样，如果你的系统支持的话，最好使用非阻塞式 I/O。

这一章里我们学习了组件、事件循环、定时器和事件联编等内容。下面的两章将会将所有的概念合在一起，应用在几个实际的问题当中。这些问题还将使我们有以超出一般普通试验的水准，来应用一下 Tk 殿堂中的两个优秀的组件：`canvas` 和 `text`。

## 相关资源

1. Tcl/Tk 库及资料。

可在地址 <http://www.sunlabs.com/research/Tcl> 中查找 Sun 的 Tcl/Tk 页面;  
从 <http://www.neosoft.com> 查找所有与 Tcl/Tk 有关的内容。

2. Perl/Tk 文档。

所有组件都有丰富的文档查找文件 [index.html](#)。

3. About Face: The Principles of User-Interface Design. Alan Cooper. IDG Books Worldwide, 1995。

有力而见解独到的有关如何设计 GUI 的著作。

4. Bringing Design to Software. Terry Winograd. Addison-Wesley, 1996。

不同领域的专家共同探讨如何进行良好的设计，尤其是用户界面设计的问题。





本章简介:

- 有关 Tetris 的介绍
- 设计
- 实现

# 第十五章

## GUI 实例： Tetris

— 这是个碰运气的游戏吗？

— 我玩的就不是这样，不是这样。

—— W.C. Fields

编写一个游戏程序是测试你对 GUI 编程理解的最好方式，因为它覆盖了有关用户界面三个重要的方面：窗体、结构化图形和动画。在这一章，我们将要编写一种流行的游戏 Tetris，完成后它将会带给我们长时间的放松与快乐。下面是我们要在本章练习使用的 Tk 知识要点：

- 使用 canvas 标签来高效的移动或删除成组的 canvas 元素 (item)。
- 使用定时器来控制动画：移动方块和发射方块。(对了，我们还要增加一种流行的街机风格的方块发射动作。)
- 使用 pack 来高效的完成窗体布局。本章只使用两个按钮和一个 canvas 组件，因此这只是一个小小的练习。

据估计 (或经常被引用的说法)，大多数带有用户界面的应用程序，要在与 GUI 相关的细节上投入 70% 的代码量。在这一章中，你将能够看到，即便是对于游戏这样的需要大量 GUI 处理的应用，Tk 是如何将工作量缩减至 30% 的。

## 有关 Tetris 的介绍

Tetris是在1985年,由它的创作者Alexey Paszhitnov、Dmitry Pavlovsky和 Vadim Gerasimov,将该游戏移植到IBM PC上时,进入PC世界的。从那不久,任天堂公司就在它的Gameboy系列手持计算机游戏中实现了它,并获得了巨大的成功。当任天堂开始在市场上推广其64位系统时,那个系列的产品仍在销售,该游戏的魅力可见一斑。

如果你从没有玩过这个游戏,我建议你玩它几次以找到一点感觉(注1)。在每次循环中,一个方块(注2)会从顶部落下(每次一行的往下移动),最终与底部的方块堆合并(如图15-1)。这时,Tetris将会消去所有排满方块的行(在堆中)。也就是将那一行删去并把堆中的所有行向下移动一行。接着下一次循环开始,这一次可能落下的是另一种不同形状的方块。这个游戏的目标就是阻止堆一直往上堆积到顶部。为了达到这种目标,你可以将方块向左和向右移动(使用键“j”和“l”)并进行旋转(使用键“k”)以使它落下时能够将下面的行填满并不停的将其消去。如果你按下空格键,方块立刻径直落到堆中(而不是一步一行的渐进下落)并与堆合并到一起。

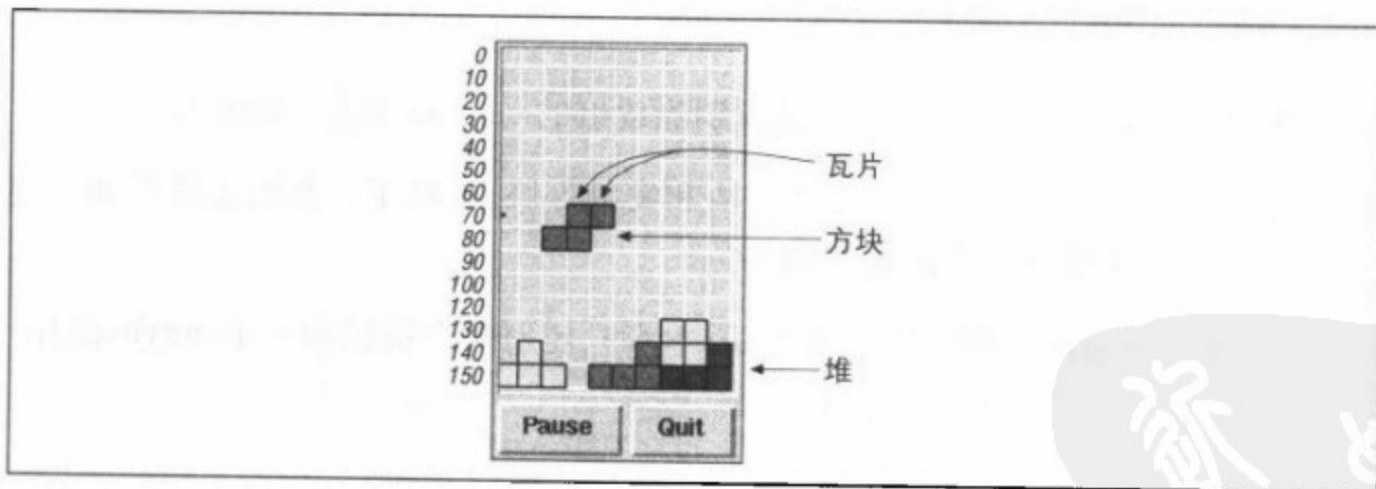


图 15-1 Tetris 的屏幕显示

注 1: 在 O'Reilly 的 FTP 站点上可以获得完整的代码, 文件名为 tetris.pl。

注 2: 由四个瓦片组成, 该游戏因而得名。

## 设计

让我们来看一下用户界面的设计（特别是我们高效使用画板组件的策略）和记录游戏状态的数据结构，它们是不依赖于用户界面的。

### 用户界面

Tetris 的屏幕布局如图 15-1 所示并不复杂。你需要两个按钮组件来分别表示“Start/Pause”和“Quit”，一个画板组件用于制图，还有一个主窗口将所有这些东西都装进去。

网格和方块将在画板上画出。每个方块由几个四方的瓦片组成，并作为一个整体进行移动。堆也类似的由一组瓦片组成。每个瓦片是一个画板元素。另一种绘制方块的方法就是使用填充多边形，但是这种由瓦片组成的版本实现起来更简单一些。方块的瓦片由字符串“block”进行标记，因此通过使用画板的 move 方法将它们作为一个单位来移动非常简单。我们将会记住每一个瓦片的所在的画板元素，这样它们就可以被单独的删除或移动。

我们对动画担心的一点，就是当显示器周期性刷新正在改变的显示内存时，所导致的闪烁。为了防止这种闪烁的发生，你需要使用双缓冲技术：就是先将新的动画帧在一张像素图绘出，然后将其迅速的拷贝到显示内存中。幸运的是，画板组件已经做了双缓冲的工作，因此我们只管将画板元素来回移动而不用担心闪烁的问题。

### 数据结构

方块和堆都要保存其瓦片的两类信息：它在网格中的位置（单元格）和由画板组件提供的 ID 值。位置可以使用行与列来表示，或是表示成单元索引如（行 \* 网格的列数 + 列）。

这个公式假定行与列均是从 0 开始的。图 15-1 描述了赋值给每行最左侧单元格的索引值。这种方案非常方便，因为它将行与列编码为一个数字，而且能够很容易的集成到我们下面要讨论的数据结构中。

我的第一个打算就是把方块和堆抽象为对象。一个瓦片有两个属性，*tile\_id*（由画板给定的ID）和*position*，而方块和堆则分别具有一个属性，也就是一个瓦片序列。但是由于任意时刻只有一个方块和一个堆，因此我选择了一种更为简单的方法。方块由数组@block\_cells表示，它的每个元素包含有被相应瓦片占据的单元格编号。类似的，数组@tile\_ids中的每个元素包含有表示该位置方块瓦片的画板组件的元素ID。堆则以另一种不同的方式进行组织。数组@heap包含了与网格中单元个数相等数目的元素；数组中的元素要么为undef，要么包含有堆中瓦片相应的画板元素ID。在我尝试的各种不同的组织方块与堆信息的方法中，我发现这种模式对于完成两种复杂的操作：方块的旋转和方块与堆的合并来说最为方便。

## 实现

我们将只关注那些对游戏来说处于核心位置的和能够演示Tk的应用的例程。

主程序只包含了init()和MainLoop()两个调用。init创建屏幕显示，建立按键绑定并配置一个定时器来调用tick子例程。让我们先从这个过程来开始实际内容的讲解。

tick把自己设置为回调函数，将方块向下移动并重新设置定时器：

```
sub tick {
    return if ($state == $PAUSED);
    if (!@block_cells) {
        if (!create_random_block()) {
            game_over();          # 堆已经满了，下一步则无法放置方块
            return;
        }
        $w_top->after($interval, \&tick);
        return;
    }
    move_down();                  # 方块下移
    $w_top->after($interval, \&tick); # 为下一个重新加载定时器
}
```

当空格键被按下时，fall()将被调用；它将使方块一直向下移动直至碰到堆中

的任何一个瓦片或是接触到底部为止。move\_down()在碰到这两种情况之一时将会返回一个 false 值。

```

sub fall {
    return if (!@block_cells);
    1 while (move_down());
}
# 敲击空格键时调用
# 如果没有初始化就返回
# 一直向下移动,直到碰到堆或底部为止

```

move\_down()只是简单的将 \$MAX\_COLS 与每一个方块瓦片的单元格位置相加,来有效的将其向下移动一行。然后它检查在这些新位置中是否有接触到网格底部,或与预先存在的堆瓦片相交的情况出现。如果有,它就会调用 merge\_block\_and\_heap 例程并返回 false 值。如果没有,它就会简单的记下这一组新的位置,并使用 move 方法将所有的方块瓦片一起向下移动(向下 \$TILE\_HEIGHT 个像素)。它返回值 1,这样其上的 fall 调用将会知道需要继续往下移动。

```

sub move_down {
    my $cell;
    my $first_cell_last_row = ($MAX_ROWS-1)*$MAX_COLS;
    # 如果已经处于堆的底部,或是下移时与堆发生了相交
    # 那么就进行合并
    foreach $cell (@block_cells) {
        if (($cell >= $first_cell_last_row) ||
            ($heap[$cell+$MAX_COLS])) {
            merge_block_and_heap();
            return 0;
        }
    }
    foreach $cell (@block_cells) {
        $cell += $MAX_COLS;
    }
    $w_heap->move('block', 0, $TILE_HEIGHT);
    return 1;
}

```

merge\_block\_and\_heap完成两个主要的任务:它将方块的所有瓦片移交给堆,并清空@block\_cells。然后它将会巡视整个@heap数组来寻找所有列中都包含有瓦片的行。如果找到了,它就会使用 addtag 方法将所有这些瓦片以另外一个名为 delete 的标签进行标记:

```
$w_canvas->addtag('delete', 'withtag' => $heap[$i]);
```

删除一行最直接的方式就是删除堆中的相应元素与画板中的相应瓦片。但是那种方法并不能让用户知道那些行被删除了；而且它还太枯燥。因此 `merge_block_and_heap` 将所有标记为 `delete` 的瓦片以白色背景填充，并在过去 300 毫秒之后再删除所有这些元素。这样用户将会看到一个整行的颜色在短暂的时间里变为白色，然后就消失了。留意一下提供给 `after` 的闭包是如何方便的提供给将来要执行的代码的。同样的闭包还将剩余的堆瓦片移动到它们新的位置（因为一些行已经被删除了）。

```
my $last_cell = $MAX_COLS * $MAX_ROWS;
sub merge_block_and_heap {
    my $cell;
    # 方块合并
    foreach $cell (@block_cells) {
        $heap[$cell] = shift @tile_ids;
    }
    $w_heap->dtag('block'); # Forget about the block - it is now merged
    # 检查填满的行，并将其清除
    # 其上所有的行都需要向下移动，包括位于 @heap 以及
    # canvas, $w_heap 中的
    my $last_cell = $MAX_ROWS * $MAX_COLS;
    my $filled_cell_count;
    my $rows_to_be_deleted = 0;
    my $i;
    for ($cell = 0; $cell < $last_cell; ) {
        $filled_cell_count = 0;
        my $first_cell_in_row = $cell;
        for ($i = 0; $i < $MAX_COLS; $i++) {
            $filled_cell_count++ if ($heap[$cell++]);
        }
        if ($filled_cell_count == $MAX_COLS) {
            # 这一行满了
            for ($i = $first_cell_in_row; $i < $cell; $i++) {
                $w_heap->addtag('delete', 'withtag' => $heap[$i]);
            }
            splice(@heap, $first_cell_in_row, $MAX_COLS);
            unshift (@heap, (undef) x $MAX_COLS);
            $rows_to_be_deleted = 1;
        }
    }
}
```

```

@block_cells = ();
@tile_ids = ();
if ($rows_to_be_deleted) {
    $w_heap->itemconfigure('delete',
                           '-fill'=> 'white');
    $w_top->after (300,
                  sub {
                      $w_heap->delete('delete');
                      my ($i);
                      my $last = $MAX_COLS * $MAX_ROWS;
                      for ($i = 0; $i < $last; $i++) {
                          next if !$heap[$i];
                          # 定位
                          my $col = $i % $MAX_COLS;
                          my $row = int($i / $MAX_COLS);
                          $w_heap->coords(
                              $heap[$i],
                              $col * $TILE_WIDTH,      #x0
                              $row * $TILE_HEIGHT,      #y0
                              ($col+1) * $TILE_WIDTH,  #x1
                              ($row+1) * $TILE_HEIGHT); #y1
                      }
                  });
}
}

```

让我们再来看看其他两个用于操纵方块的例程: `move_left` 和 `rotate`。我们将跳过 `move_right`, 因为它与 `move_left` 类似。

`move_left` 通过将其相应单元格的位置减1来把方块的每个瓦片向左移一下。如果新位置会超出左边的边界, 或与一个已经被占用的堆单元相交, 那么该函数会将什么也不做。

如果允许移动的话, 画板上标记为 “block” 的元素将会简单的向左移动 `$TILE_WIDTH` 个像素:

```

sub move_left {
    my $cell;
    foreach $cell (@block_cells) {
        # 检查单元是否已经处于最左边了
        # 如果没有, 就检查单元的左边的位置是否已经
    }
}

```

```

    # 被占了
    if ((( $cell % $MAX_COLS) == 0) ||
        ($heap[$cell-1])){
        return;
    }
}
foreach $cell (@block_cells) {
    $cell--; # 这会影响 @block_cells的内容
}

$w_heap->move('block', - $TILE_WIDTH, 0);
}

```

rotate 稍微有点复杂。它根据方块瓦片的位置计算出一个行列坐标轴, 并使用一种下面将要讲到的简单变换计算出新的瓦片位置。它还确保这种新计算出来的位置不会存在任何的不合法性(如超出了网格空间或者与堆相交)。它然后将会调用画板的 coords 方法来将每个单一的瓦片移动到新的位置。

```

sub rotate {
    # 逆时针旋转方块
    return if (!@block_cells);
    my $cell;
    # 计算旋转轴心
    # 轴心位于 所有方块单元的(平均 x, 平均 y) 位置
    my $row_total = 0; my $col_total = 0;
    my ($row, $col);
    my @cols = map {$_ % $MAX_COLS} @block_cells;
    my @rows = map {int($_ / $MAX_COLS)} @block_cells;
    foreach (0 .. $#cols) {
        $row_total += $rows[$_];
        $col_total += $cols[$_];
    }
    my $pivot_row = int ($row_total / @cols + 0.5); # 轴心行
    my $pivot_col = int ($col_total / @cols + 0.5); # 轴心列
    # 要想逆时针旋转定位每个单元, 需要做一些小的变换
    # 从轴心的行偏移量转变为等价的列偏移量
    # 而列偏移量转变为反方向的行偏移量
    my @new_cells = ();
    my @new_rows = ();
    my @new_cols = ();
    my ($new_row, $new_col);
    while (@rows) {
        $row = shift @rows;

```



```

    $col = shift @cols;
    # 计算新的 $row 和 $col
    $new_col = $pivot_col + ($row - $pivot_row);
    $new_row = $pivot_row - ($col - $pivot_col);
    $cell = $new_row * $MAX_COLS + $new_col;
    # 检查这些新的行和列是否有效 (是否出界了或是
    # 那个单元已经被占了)
    # 如果有效, 那么就不应东西可以占了
    if (($new_row < 0) || ($new_row > $MAX_ROWS) ||
        ($new_col < 0) || ($new_col > $MAX_COLS) ||
        $heap[$cell]) {
        return 0;
    }
    push (@new_rows, $new_row);
    push (@new_cols, $new_col);
    push (@new_cells, $cell);
}
# 将UI 瓦片移动到合适的坐标
my $i = @new_rows-1;
while ($i >= 0) {
    $new_row = $new_rows[$i];
    $new_col = $new_cols[$i];
    $w_heap->coords($tile_ids[$i],
                    $new_col * $TILE_WIDTH,      #x0
                    $new_row * $TILE_HEIGHT,     #y0
                    ($new_col+1) * $TILE_WIDTH,  #x1
                    ($new_row+1) * $TILE_HEIGHT);
    $i--;
}
@block_cells = @new_cells;
1; # 成功
}

```

在这个变种版本的 Tetris 开始执行时, 它画出了一个红色的小三角用以表示“大炮”(如图 15-1 中单元号为 70 的那一行)。每当按下“a”或“s”时, shoot 子例程将被调用。当方块碰巧经过大炮所在行时, 按键“a”将会使枪发射出一个箭头并将方块最左边的瓦片炸掉。而按键“s”则将向最右边的瓦片射击。这的确太土了, 不过如果你想看一下如何使用画板来实现动画序列的话, 还是挺有用的。过程的第一部分只是判定, 如果有的话将要移去那一个方块瓦片。它然后创建一个从大炮到选中瓦片的箭头, 改变火花的形态, 然后在过 200 毫秒后, 将箭头与瓦片同时删去。这就是炸掉一个瓦片的视觉效果。

```

sub shoot {
    my ($dir) = @_;
    my $first_cell_shoot_row = $shoot_row*$MAX_COLS;
    my $last_cell_shoot_row = $first_cell_shoot_row + $MAX_COLS;
    my $cell;
    my (@indices) =
        sort {
            $dir eq 'left'?
                $block_cells[$a] <=> $block_cells[$b] :
                $block_cells[$b] <=> $block_cells[$a]
            } (0 .. $#block_cells);
    my $found = -1;
    my $i;
    foreach $i (@indices) {
        $cell = $block_cells[$i];
        if (($cell >= $first_cell_shoot_row) &&
            ($cell < $last_cell_shoot_row)) {
            $found = $i;
            last;
        }
    }
    if ($found != -1) {
        my $shot_tile = $tile_ids[$found];
        ($cell) = splice (@block_cells, $found, 1);
        splice (@tile_ids, $found, 1);
        my $y = ($shoot_row + 0.5)*$TILE_HEIGHT;
        my $arrow = $w_heap->create(
            'line',
            0,
            $y,
            (($cell % $MAX_COLS) + 0.5)
                * $TILE_WIDTH,
            $y,
            '-fill' => 'white',
            '-arrow' => 'last',
            '-arrowshape' => [7,7,3]
        );
        $w_heap->itemconfigure($shot_tile,
            '-stipple' => 'gray25');
        $w_top->after (200,sub {
            $w_heap->delete($shot_tile);
            $w_heap->delete($arrow);
        });
    }
}

```

```
    }  
}
```

让我们现在来看一下负责设置屏幕的两个例程: `create_screen` 与 `bind_key`。这两个函数均由 `init()` 调用。请留意一下 `pack` 方法如何在 `create_screen` 中使用, 以及空格符在 `bind_key` 中如何转换成一个事件编联。

```
sub create_screen {  
    $w_top = MainWindow->new(-title => 'Tetris - Perl/Tk');  
    $w_heap = $w_top->Canvas('-width' => $MAX_COLS * $TILE_WIDTH,  
                             '-height' => $MAX_ROWS * $TILE_HEIGHT,  
                             '-border' => 1,  
                             '-relief' => 'ridge');  
    $w_start = $w_top->Button('-text' => 'Start',  
                              '-command' => \&start_pause,  
                              );  
    my $w_quit = $w_top->Button('-text' => 'Quit',  
                                '-command' => sub {exit(0)}  
                                );  
    $w_heap->pack();  
    $w_start->pack('-side'=> 'left', '-fill' => 'y', '-expand' => 'y');  
    $w_quit->pack('-side'=> 'right', '-fill' => 'y', '-expand' => 'y');  
}  
sub bind_key {  
    my ($keychar, $callback) = @_;  
    if ($keychar eq '') {  
        $keychar = "KeyPress-space";  
    }  
    $w_top->bind("<${keychar}>", $callback);  
}
```

PDF  
PDG

## 第十六章

# GUI 实例：

# Man 页面查看器

本章简介：

- man 与 perlman
- 实现
- 相关资源

招收心灵感应者。你该知道到哪里去应聘。

—— 无名氏

本章的首要目的就是练习使用Tk文本框组件的一些重要功能；一个名为perlman的man页面查看器提供了一个理想的训练案例（注1）。在这一章你将会熟悉该组件的文本插入、删除与检索功能；使用多种多样的索引原语；创建与配置标签；并进行正则表达式的搜索与文本的加强显示。同时你将还会学习使用输入条组件和动态的构造菜单。

perlman 参照的原型为Tkman，这是一个非常好的man页面查看器。它是由加利福尼亚大学伯克利分校的Thomas Phelps利用Tcl/Tk来编写的。perlman只包含了TkMan的一小部分功能。

Thomas还写了一篇名为《Two Years with TkMan: Lessons and Innovations. Or, Everything I Needed to Know about Tcl/Tk I Learned from TkMan (TkMan的两年：教训与创新，或称，我从TkMan中学到的有关Tcl/Tk的一切)》的体验文章。他列举了使用一种脚本语言来编写整个工具的良好案例，而无意中却使我们这些沾沾自喜的Perl程序员确信，为什么Perl本应当是理想的选择。

注1： 想要使用文本框组件进行更具挑战的试验的话，你可以试着编写一个HTML页面查看器。

## Man 与 perlman

Unix 的手册页面通常使用 `man(1)` 命令来查看。当你在命令行下输入 `man perl` 时，它将会搜索由环境变量 `MANPATH` 所指示的一组路径，`MANPATH` 中包含有由冒号分隔的路径名。（如果 `MANPATH` 没有定义，`man(1)` 则查看标准路径，如 `/usr/man`。）一旦它找到了一个名为 `perl.1` 的文件，它就会调用 `tbl` 来格式化图表以及 `nroff` 来格式化文本，并以管道的形式输出到一个合适的分页查看器，如 `more(1)` 或 `less(1)` 中。

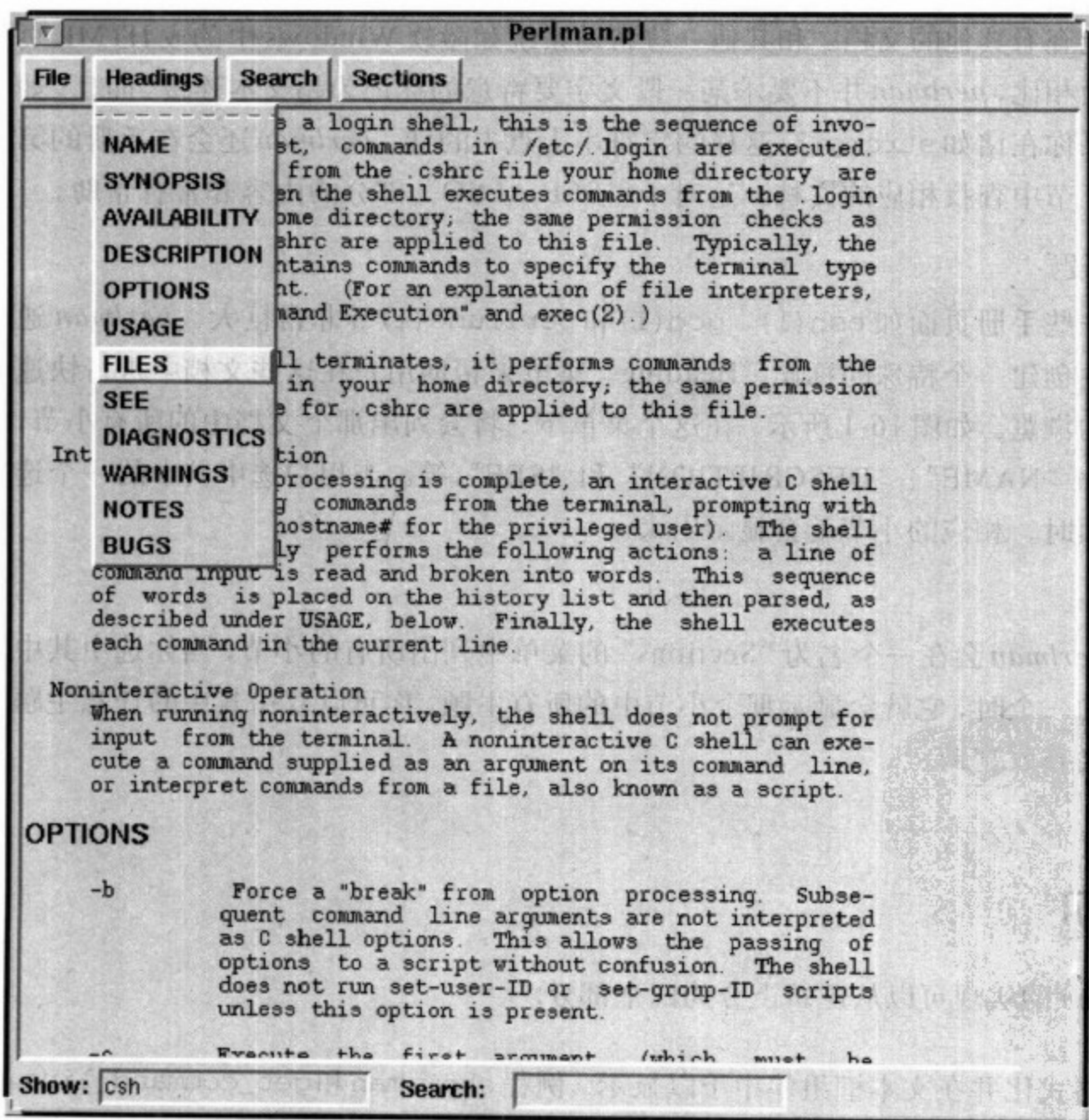


图 16-1 perlman 的屏幕显示

如图 16-1 所示, *perlman* 就是一个对 `man` 命令的 GUI 包裹程序。要想查看一个主题, 你必须在“Show”输入框中输入一个字符串并敲回车。要想加重显示所有匹配一个给定正则表达式的文本, 只需简单的“Search”输入框中输入这个字符串并敲回车。你还可以进行精确的字符串搜索并可以有选择的忽略大小写。此外, *perlman* 还提供了下面的一些特性:

### 超文本

如果你在任意的单词上双击, 如果它存在的话, *perlman* 就会显示与那个主题相关的 `man` 页面。而在 `man(1)` 命令中, 你必须退出当前的 `man` 页面才能查看其他的文档。和其他一些帮助系统如微软 Windows 中的或 HTML 页面相比, *perlman* 并不要求某一段文字要特意的标记为超文本连接。而且, 如果你在诸如 `strcmp(3)` 这样的字符串上点击的话, *perlman* 还会在手册的第三节中查找相应的文档。这对于“SEE ALSO”部分的内容非常有帮助。

### 小节标题

一些手册页面如 `csh(1)`、`gcc(1)` 和 `perlfunc(1)` 等非常巨大。 *perlman* 通过创建一个特殊的称作“Heading”菜单来帮助用户在这些文档中进行快速的浏览。如图 16-1 所示, 在这个菜单下, 将会列出那个文档中的所有小节, 如“NAME”、“DESCRIPTION”和“SEE”等。当用户选中其中的一个选项时, 相应的小节就会显示出来。

### 小节

*perlman* 会在一个名为“Sections”的菜单中列出所有的小节, 当你选中其中的一个时, 它就会显示那个小节中的所有主题。你可以双击其中的任意主题来获取帮助。

## 实现

*perlman* 的实现可以从逻辑上分为四个部分:

- 格式化并在文本框组件中予以显示: 例程 `show_man` 和 `get_command_line`。
- 搜索机制: `search`。

- 屏幕布局: `create_ui`。
- 显示每一节中的帮助主题列表。我们不讲述这一块功能,因为它不涉及大量的界面代码。

在详细讨论上面提到的每个子例程之前,我们先来简短的学习一下由 *perlman* 所使用的文本框组件的所有功能:

- 在文本末尾进行插入操作,并以标签 (“section”) 进行标记:

```
$text->insert('end', 'sample text', 'section');
```

- 获取两个索引之间的一段文本:

```
$line = $text->get($start_index, $end_index);
```

- 使某个索引可见:

```
$text->see($index)
```

- 删除整个内容:

```
$text->delete('1.0', 'end'); # 从第一行, 0 列到末尾
```

- 创建并配置一个标签:

```
$text->tagConfigure('search',  
                    'foreground' => yellow, 'background' => 'red');
```

- 删除一个标签:

```
$text->tagDelete('search');
```

- 给定一个索引位置与字符的数量,为一段文本设置标签:

```
$text->tagAdd('search', $current, "$current + $length char");
```

- 列出所有的标记并挨个删除:

```
foreach $mark ( $text->markNames() ) { $text->markUnset($mark); }
```

- 根据逻辑位置获取行号与列号:

```
# 当前结束位置的行与列  
$index = $text->index('end');  
# 到达当前的插入位置, 然后再到单词的起始位置  
# 并报告行与列  
$start_index = $text->index('insert wordstart');
```

```
# 到第10行第3列，向后5个字符，并报告新的
# 行与列
$i = $text->index("10.3 + 5 char");
```

注意，`index`方法并不改变组件的状态。

- 进行一项精确的或正则表达式搜索，并指定从哪里开始到哪里结束：

```
$current = $text->search('-count' => \ $length,
                        '-regex', '-nocase', '--', # 搜索选项
                        $search_pattern,
                        $current, 'end');          # 起止位置
```

`search`方法将会返回搜索成功位置的索引，并设置与`-count`属性相关的变量。如果搜索失败它就会返回`undef`值。

- 将双击鼠标绑定到一个子例程：

```
$text->bind('<Double-1>', \&pick_word);
```

## Man 页面的显示

让我们现在来讨论该应用的实质性内容，子例程 `show_man`。如图 16-1 所示，一个标签为“Show:”的输入组件用以接收一个主题名称。当用户在这个组件 `$show` 中输入文本并敲击回车后，`show_man` 将被调用。这个子例程从 `$show` 中取出字符串，并调用 `get_command_line` 来构造一个可从 `man` 命令中读取的管道（为了供 `open` 使用）。它然后每次一行的从管道中读取信息，并检查它是否是一种类似于标题的东西（例如“NAME”或“DESCRIPTION”）。`man` 页面中的标题通常为全大写形式而且均从第一列开始。如果一行看起来像是一个标题，`show_man` 将会以一个名为“section”的标签插入这一行；否则它会将其作为未标记的文本进行插入。“section”标签预先配置以大一些的字体。此外，`show_man` 还在“Heading”菜单中增添一个新的条目，并安排与该条目关联的回调函数，将文本框组件滚动至包含该小节标题的那一行。

```
sub show_man {
    my $entry = $show->get(); # 从 $show 获得 entry 的内容
    # $entry 可以为 'csh', 或 'csh(1)'
    my ($man, $section) = ($entry =~ /^(w+)(\(.*\))?/);
    if ($section && (!is_valid_section($section))) {
        undef $section ;
    }
}
```



```

}
my $cmd_line = get_command_line($man, $section); # used by open

# 清除与本页有关的所有内容（内容、菜单、标记）
$text->delete('1.0', 'end'); # 清除当前页
# 插入 'working' 消息；使用标签 'section'
# 因为它有漂亮的大字体
$text->insert('end',
             "Formatting \"$man\".. please wait", 'section');
$text->update(); # 向文本框组件刷新任何改变
$menu_headings->menu()->delete(0, 'end'); # 删除当前的标题
my $mark;
foreach $mark ($text->markNames) { # 删除所有的标记
    $text->markUnset($mark);
}

# UI 现在干净了。打开文件
if (!open (F, $cmd_line)) {
    # 使用文本框组件来输出错误信息
    $text->insert('end', "\nError in running man or rman");
    $text->update();
    return;
}

# 清除消息 "Formatting $man ..."
$text->delete('1.0', 'end');
my $lines_added = 0; my $line;

while (defined($line = <F>)) {
    $lines_added = 1;
    # 如果字符为大写形式，它就有可能是一个小节
    if ($line =~ /^[A-Z]/) {
        # 有可能是一个小节头
        ($mark = $line) =~ s/\s.*$//g; # $mark 中包含有小节标题
        my $index = $text->index('end'); # 记住当前的结束位置
        # 为小节标题标记 'section' 标签
        $text->insert('end', "$mark\n\n", 'section');
        # 创建一个小节头菜单条目
        # 让回调函数调用文本框组件的方法 'see'
        # 来到达上面记下的索引位置
        $menu_headings->command(
            '-label' => $mark,
            '-command' => [sub {$text->see($_[0])}, $index])
    } else {
        $text->insert('end', $line); # 为普通文本，只管插入
    }
}

```

```

    }
}
if ( ! $lines_added ) {
    $text->insert('end', "Sorry. No information found on $man");
}
close(F);
}

```

`get_command_line`接受一个man页面的名字与一个可选的小节号为参数并返回一个可供 `open` 命令使用的相应的命令行。不同的系统可能要求不同形式的命令行，下面列举描述的是Solaris上的命令行。由于`man`是为终端格式化页面的（插入转义符序列来将一些单词加重显示并显示标题与脚注），因此我们使用了一个可自由获得的称作`rman`（“RossetaMan”；请参照本章末尾的“相关资源一节”）的工具来去除一些乱码信息。

```

sub get_command_line {
    my ($man, $section) = @_;
    if ($section) {
        $section =~ s/[()]/ /g; # remove parens
        return "man -s $section $man 2> /dev/null | rman |";
    } else {
        return "man $man 2> /dev/null | rman |";
    }
}

```

`pick_word`过程是在你双击文本框组件时被调用的。它使用`index`方法来计算出被点击单词的起始位置，以及行的末尾，并抽取这个区间的文本。`pick_word`然后在其中查找一个普通的字符串（标题），其后跟有一个可选的包裹在括号中的字符串（小节）。在调用`show_man`之前，它先将这个字符串插入到输入组件`$show`中，假装成是一个用户敲入了该字符串一样。

```

sub pick_word {
    my $start_index = $text->index('insert wordstart');
    my $end_index = $text->index('insert lineend');
    my $line = $text->get($start_index, $end_index);
    my ($page, $section) = ($line =~ /^(\w+)(\(.*\?\)\)?/);
    return unless $page;
    $show->delete('0', 'end');
    if ($section && is_valid_section($section)) {
        $show->insert('end', "$page${section}");
    }
}

```

```

    } else {
        $show->insert('end', $page);
    }
    show_man();
}

```

## 文本搜索

这里的菜单条包含了一个与我们在第十四章“使用Tk进行用户界面编程”中“菜单”一节中那个例子一样的搜索菜单。当“Find”菜单项被选中时，子例程 `search` 将会被调用。它首先调用 `tagDelete` 来删除所有的加强显示内容（这可能是前一次查找中留下的）。然后它从顶部（第1行，第0列）开始调用组件的 `search` 方法来查找第一处匹配的文本。如果找到了，该方法就会以匹配文本的长度来更新提供给参数 `-count` 的变量。这一段文本将会使用一个名为“search”的标签来进行加强显示。光标将会推进到匹配文本的后面然后在继续进行搜索。

[illegible]

```

        last if (!$current);
        # 将匹配文本区域标记为 'search'
        $text->tagAdd('search', $current, "$current + $length char");
        # 向前移动光标并继续搜索
        $current = $text->index("$current + $length char");
    }
}

```

## 屏幕布局

create\_ui 用来建立起简单的用户界面。请特别注意提供给 pack 的 padding 选项以及文本和输入组件的事件联编设置情况。

```

sub create_ui {
    my $top = MainWindow->new();
    #-----
    # 创建菜单
    #-----
    # Menu bar
    my $menu_bar = $top->Frame()->pack('-side' => 'top',
                                       '-fill' => 'x');

    #----- 文件菜单 -----
    my $menu_file = $menu_bar->Menubutton('-text' => 'File',
                                          '-relief' => 'raised',
                                          '-borderwidth' => 2,
                                          )->pack('-side' => 'left',
                                                  '-padx' => 2,
                                                  );

    $menu_file->separator();
    $menu_file->command('-label' => 'Quit',
                      '-command' => sub {exit(0)});

    #----- 小节菜单 -----
    $menu_headings = $menu_bar->Menubutton('-text' => 'Headings',
                                           '-relief' => 'raised',
                                           '-borderwidth' => 2,
                                           )->pack('-side' => 'left',
                                                  '-padx' => 2,
                                                  );

    $menu_headings->separator();
}

```

```

#-----检索菜单-----
my $search_mb = $menu_bar->Menubutton('-text'      => 'Search',
                                       '-relief'     => 'raised',
                                       '-borderwidth' => 2,
                                       )->pack('-side' => 'left',
                                              '-padx'  => 2
                                              );

$match_type = "-regexp"; $ignore_case = 1;
$search_mb->separator();

# Regexp match
$search_mb->radiobutton('-label'    => 'Regexp match',
                      '-value'     => '-regexp',
                      '-variable' => \$match_type);

# Exact match
$search_mb->radiobutton('-label'    => 'Exact match',
                      '-value'     => '-exact',
                      '-variable' => \$match_type);

$search_mb->separator();
# 忽略大小写
$search_mb->checkboxbutton('-label'    => 'Ignore case?',
                      '-variable' => \$ignore_case);

#-----小节菜单-----
my $menu_sections = $menu_bar->Menubutton('-text' => 'Sections',
                                           '-relief' => 'raised',
                                           '-borderwidth' => 2,
                                           )->pack('-side' => 'left',
                                                  '-padx' => 2,
                                                  );

# 使用 %sections 的键值来初始化小节菜单
my $section_name;
foreach $section_name (sort keys %sections) {
    $menu_sections->command (
        '-label' => "($section_name)",
        '-command' => [\&show_section_contents, $section_name]);
}

#-----
# 创建并配置文本框组件以及显示与检索输入条组件
#-----
$text = $top->Text ('-width' => 80,
                  '-height' => 40)->pack();
$text->tagConfigure('section',
                  '-font' =>

```

```
'-adobe-helvetica-bold-r-normal 4-140-75-75-p-82-iso8859-1');  
# 对这种字体规范使用 xlsfonts(1)  
$text->bind('<Double-1>', \&pick_word);  
$stop->Label('-text' => 'Show:')->pack('-side' => 'left');  
$show = $stop->Entry ('-width' => 20,  
                      )->pack('-side' => 'left');  
$show->bind('<KeyPress-Return>', \&show_man);  
  
$stop->Label('-text' => 'Search:'  
            )->pack('-side' => 'left', '-padx' => 10);  
$search = $stop->Entry ('-width' => 20,  
                       )->pack('-side' => 'left');  
$search->bind('<KeyPress-Return>', \&search);  
}
```

请查看一下与这本书的软件打包在一起的文件 *perlman.pl*。这个软件包可以从 O'Reilly 的 FTP 站点获取。如果愿意的话,你也可以为这个工具增添几处有用(而简单)的功能,如:为格式化过的页面提供高速缓冲以及显示一个给定主题名的所有 man 页面(而不只是 MANPATH 中所碰到的第一个页面)。

## 相关资源

1. TkMan and RosettaMan (rman). Thomas Phelps.

下载地址为: <ftp://ftp.cs.berkeley.edu/pub/people/phelps/tcl>。

2. Two Years with TkMan: Lessons and Innovations. Or, Everything I Needed to Know about Tcl/Tk I learned from TKMan. Thomas Phelps.

有趣的研究 Tcl/Tk 用法的案例,请查看上面的地址。

资源分享网  
PDG

本章简介：

- 有关代码生成的问题
- Jeeves 实例
- Jeeves 概述
- Jeeves 的实现
- 规格语法分析器样例
- 相关资源

# 第十七章

## 模板驱动的 代码生成

我宁可编写程序去编写程序，  
也不愿意编写程序。(译注1)

——Jon Bentley 《Programming Pearls》

本章将构建一个模板驱动的代码生成器，这种软件是C、C++或Java程序员工具箱中不可或缺的一种工具。本章有两个目的：一是阐明作为一种代码重用方法的代码生成问题，然后是提出一种并不简单的问题，让大家练习一下我们在前半本书中所学到的所有概念：复杂数据结构、模块、对象以及eval。请大家尽情的享用Perl吧！

## 有关代码生成的问题

程序员们总是在不停的创建与使用小巧的规格语言。数据库模式，资源（在Unix中的rc文件，如.mwmrc和.openwinrc），用户界面规格（Motif的UIL文件），网络接口规格（RPC或CORBA IDL文件）等等均为此类语言的范例。这种语言可以使你以一种高级、精简及陈述性的格式来描述你的需求。例如，在Motif的UIL（用户接口语言）中，你只需简单的陈述一下你需要放在一个窗体中的两个按钮，就可以达到编写20行左右C语句的效果。

译注1：原文为“I'd rather write programs to write programs than write programs”。译文很难体现原文的韵味。

这种规格语言与传统系统编程语言如C或C++之间存在的巨大的语义性差异,可以通过两种方式来进行沟通。第一种就是使C应用程序将这种规格说明看成是一种元数据;也就是说,该应用程序内嵌了规格语法分析器,并使用C数据结构和一种内部API来与它交换信息。另一种方法就是使用一个独立的编译器将这种规格说明翻译成C语言,并将其与应用进行连接。RPC系统与CASE工具则更倾向于使用这种方式。

在下面的章节中,我们将学习第二种方式并亲手构建一个可配置的代码生成框架,取名为Jeeves (注1)。

我们前面提到的代码生成器很显然是特定于某一领域的。实际应用中我还发现它们的输出能力有着不必要的专有性。考虑下面的例子:

### *RPC*

**注意:** RPC机制允许你调用不同地址空间,甚至是不同机器中的过程。你使用一种接口定义语言(IDL)来说明你要输出的过程,并将其作为IDL编译器的输入,于是该编译器将会为服务器及客户端产生一些C代码。将这些代码与你的应用程序的代码进行连接,然后你就会拥有网络透明的过程调用了。

大多数的商用IDL编译器在改变它们的输出代码方面相当的不灵活。它们使你很难插入代码用于网络性能监控或是检查网络中的数据流。如果你需要在发送到网络上之前进行透明的数据加密,那么就会运气不佳。当然,你可以改变由IDL编译器输出后的代码,但是当你下次在运行IDL编译器时这种改动会被覆盖。

### *CASE*

许多CASE工具从对象模型规格说明中产生C代码。下面的样例规格说明列举了实体类和它们的属性,并指出了这些类之间关联的程度(degree)和势(cardinality):

```
Employee {  
    int          emp_id    key  
    string[40]   name  
    Department   dept_id
```

注1: Jeeves是P.G. Wodehouse小说中能干的男管家,他只要动一下眉毛就会替他的笨主人做完所有的活儿。



```

        double      salary
    }
    Department {
        int          dept_id  key
        string[20]   name
    }
    Relationship Department(1) contains Employee (n)

```

给定了这样一种小巧的规格说明语言, 举例来说, 我们可以自动的生成C和嵌入式SQL代码来进行数据库表的维护, 如下所示:

```

int create_employee_table {
    exec sql create table employee_table (
        employee_id integer,
        name varchar, salary float);
    return check_db_error();
}
int create_employee (employee *e) {
    if (!check_dept(e->dept))
        return 0;
    e->employee_id = ++g_employee_id;
    exec sql insert into table employee_table (
        employee_id, name, salary)
        values (:*e);
    return check_db_error();
}

```

这个规格说明还提供了足够的信息, 说明如何生成用于每个实体和管理参考完整性约束 (“如果一个部门对象中包含有一个或多个雇员, 那么该部门对象不能被删除”) 的C++ 类的代码。

大多数CASE工具都局限于只能生成一种代码模式。如果将来你要购买面向对象数据库, 那么早先的输出代码将变得用处不大。要是此模式是硬编码的, 你就只剩下一个图表工具了 (也是极贵的一个)。

### POD、Javadoc

整个Perl文档都是以一种称做POD (plain old documentation, 普通旧式文档) 格式来编写的。它提供了一种简单而高级的原语用来指定段落风格 (=head1、=item) 和字符的风格 (例如, B<foo> 将会以黑体打印出这个词)。发行版中包含有一些转换工具如 *pod2text*、*pod2html*、*pod2man* 等等。POD文档可以嵌入在代码中, 并由这些工具来抽取 (Perl解释器将会忽略这

些格式命令)。由于文档与代码是集成在一起的,因此这种机制降低了代码与文档不匹配的可能性。

与之类似,所有的Java库都使用一种称做Javadoc的格式来建立文档。这种文档可以使用一个名为javadoc的工具来加以提取或转换成HTML格式。

这两种工具集都受限于特定的输出(ASCII、HTML等)。例如,如果你想编写一个pod2rt转换器(这是在微软Windows系统中使用的Rich Text Format格式的文档),你就得从头开始,因为POD语法分析器不是以单独软件包的方式提供的。一种更好的选择应当是以POD语法分析器为中心,然后编写几种不同类型即插即用的后端模块。

### SWIG、XS

在第十八章“扩展Perl:第一课”中,我们将会有机会学到两个名为SWIG和XS的工具。给定一个用户界面规格说明,它们就会产生能够将Perl与定制的C扩展联编起来的代码。实际上,SWIG就是一种我们将要构建的经典的代码生成类型的例子:由于其后端是模板驱动的,因此它能够从一种规格说明语言产生各种类型的输出代码。

在大多数情况下,对多种类型输出的需求,通常要超过对输入规格的改变次数。这种现象将会衍生出两种结论。首先,对输入进行的语法分析与产生最终的输出是相关但又分立的任务。第二,输出应当是可配置的。我们要么拥有参数化的输出生成器,要么就是有许多可以同输入语法分析器互换使用的输出生成器。就我的经验而言,第一种选项经常不那么实用。例如,要为POD编写一个输出生成器,使它在改动几个参数后就能够输出HTML、ASCII或RTF格式的信息。这实在没有什么意义,因为这些输出集是那么的不同的。

Jeeves框架采取第二种方案。通过提供一个模板驱动的代码生成后端,它就可以帮助你编写一个可配置的翻译器。这个模块允许你编写带有循环语句,if/then条件测试,变量,以及部分Perl代码的可配置的模板,因此这可不是一种普通的产生工具的工具。(不然,它也许就应该被称做yacccc了。)

让我们通过例子来更好的解释这个框架。

## Jeeves 的例子

我们先来看一个非常简单的对象模型规格文件，其中包含了一组类，每个类又包含一组类型属性：

```
// 文件: emp.om (om 代表对象模型)
class Employee {
    int      id;
    string    name;
    int      dept_id;
};

class Department {
    int      id;
    string    name;
};
```

我们想根据这个规格说明来为每个类产生一个 C++ 头文件。例如，假设文件 *Employee.h* 将会是下面的这个样子 (*Department.h* 与之类似)：

```
#ifndef _Employee_h_
#define _Employee_h_
#include <Object.h>
// 文件 : 'Employee.h'
// 用户 : "sriram"
class Employee : Object {
    int id;
    string name;
    int dept_id;
    Employee(); // 私有构造函数，使用 Create()
public:
    // 方法
    static Employee* Create();
    ~Employee();
    // 存取方法;
    int  get_id();
    void set_id(int);
    string  get_name();
    void set_name(string);
    int  get_dept_id();
    void set_dept_id(int);
}
#endif
```



我们将使用Jeeves而不是屈就于编写一次性脚本程序的诱惑，来处理这种特殊的工作。这种方案有三个步骤的工作要做：

1. 为对象规格编写一个语法分析器模块。
2. 编写一个模板来创建所需要的输出。
3. 以规格语法分析器的名字，模范文件，以及实例规格文件的形式来调用Jeeves。

这种方案迫使你把语法分析与输出阶段分成两个不同的模块。你也许觉得编写一次性的脚本程序更简单些，但是这可不是真的如此：你仍然需要解决规格语法分析与产生输出的问题。如果你使用Jeeves的方式，你还可以利用它的模板处理机制。Jeeves要求语法分析器将规格说明概括为一种称做抽象语法树（AST）的数据结构；Jeeves并不帮助你进行语法分析；毕竟，它怎么会了解我们前不久随意编制出的一种语言呢？

如图17-1所示，语法树就是对属性和属性列表的一种简单的树形排列。带阴影的方框表示AST节点，而外层的盒子表示这些节点的集合（矢量属性）。语法树中的每个节点都有一个或多个属性（或名-值对）。属性值可以是一个标量变量（`class_name`, `attr_name`, `attr_type`）也可以是一个包含其他节点的矢量（`attr_list`和`class_list`）。就当前的实现而言，Jeeves并不要求节点包含任何其他类型的值（如指向其他类型类型数组或散列表的引用）。

为了对Jeeves的工作流程有一个快速的了解，我们现在将假定输入规格语法分析器已经编写完毕并能够产生如图17-1中所示的语法树。我们将在后面“规格语法分析器样例”一节中解释它的实现。

下一步我们要编写一个模板文件（名为`oo.tpl`）来输出所要求的文件。Jeeves允许你把语法树中的属性当成变量来使用，而且提供关键词来循环操纵矢量属性。例17-1中的模板一次产生两个文件。

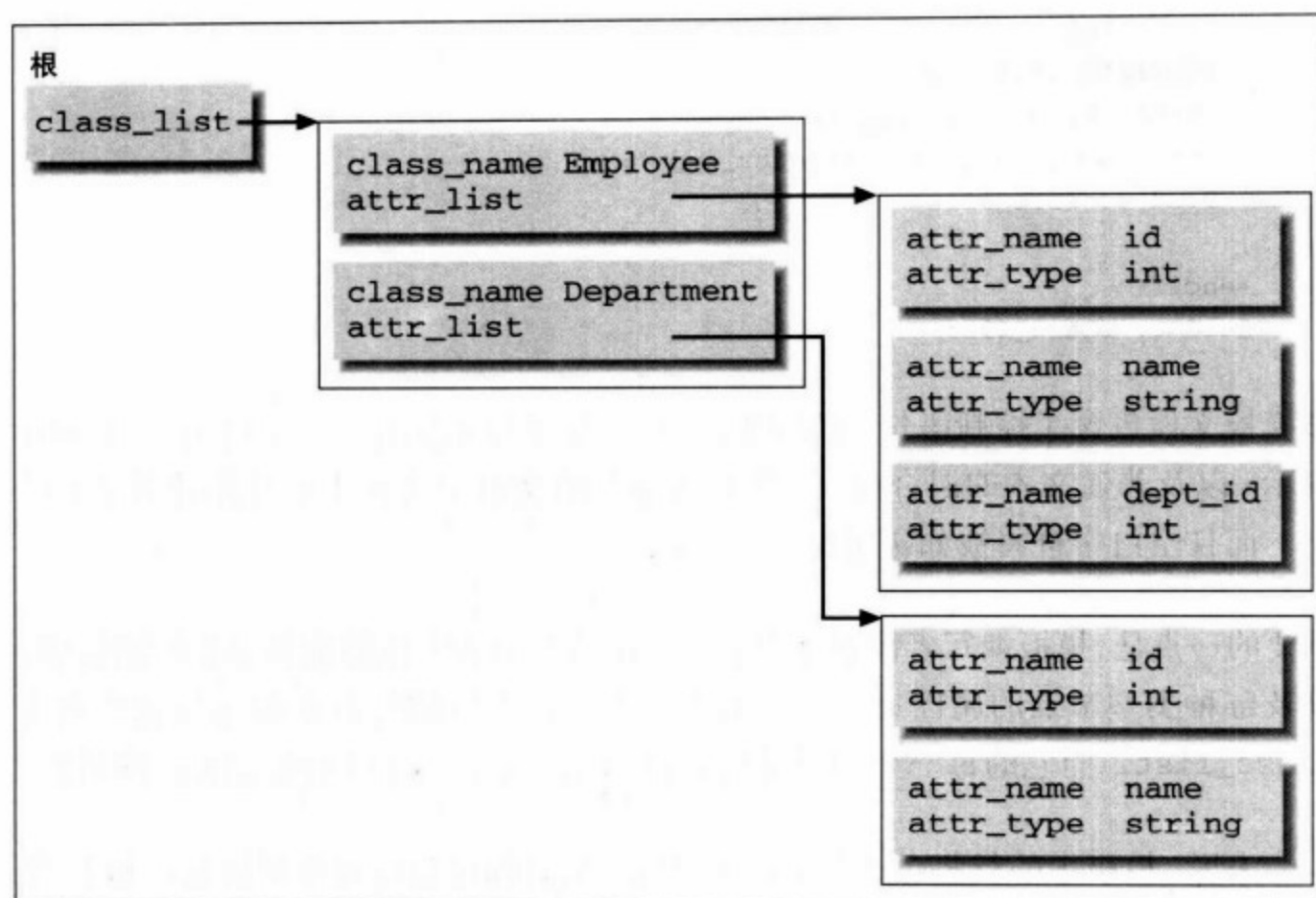


图 17-1 语法树示例

例 17-1: oo.tpl: 模板文件

```

@foreach class_list
@//-----
@// 注意：我们在上面的 foreach 中打开了一个新的 ".h" 文件 ...
  @openfile ${class_name}.h
  #ifndef _${class_name}_h_
  #define _${class_name}_h_
  #include <Object.h>
  @perl $user = $ENV{"USER"};
  // 文件：'${class_name}.h'
  // 用户："$user"
  class $class_name : Object {
    @foreach attr_list
      $attr_type $attr_name;
    @end
    $class_name(); // 私有构造函数，使用 Create()
  public:
    // 方法
    $class_name* Create();
    ~$class_name();
  
```

```

    // 存取方法;
    @foreach attr_list
        $attr_type    get_${attr_name}();
        void set_${attr_name}($attr_type);
    @end .. attr_list
}
#endif
@end .. class_list

```

该模板文件包含了控制语句(那些以@开始并加重显示的行),属性(由\$varname表示)以及普通文本的混合体。那些加重显示的文本将会在变量替换后简单的输出,而且空白字符将被如实保留下来。

重要的一点是,模板编写者需要知道由输入语法分析器产生的那种类型的语法树,以及每种类型节点的属性集。在前面的例子中,模板编写者必须知道类列表为class\_list,而它的每一个元素又具有诸如class\_name和attr\_list的属性。

一般来说,模板中的行将会进行简单的替换(所有标量变量将在原地被扩展),然后写入由上次@openfile结构打开的文件中。如果行位于一个@foreach...@end代码块中,那么它将被替换与写出好几次。@foreach代码块在语法树中的一个列表值的属性中进行迭代,并使当前AST节点的属性成为全局变量。例如,@foreach class\_list将会“访问”由class\_list的属性所指向的每一个节点,并使变量\$class\_name与\$attr\_list(请参见图17-1)对于跟在@foreach指令后的文本可用。在前面提到的样例模板中,由于@open\_file就位于这样一个代码块中,并使用\$class\_name作为文件名,因此模板将在每次迭代中生成一个新文件。普通的模板行将会被简单的输出到当前打开的文件中。@perl命令允许你在内建的原语不能够胜任时嵌入Perl代码。在讨论模板语法分析器的实现时,我们将会了解一些更多的模板指令。

在编写完了对象模型规格语法分析器,OO\_Schema.pm,模板oo.tpl以及我们的样例规格说明之后,我们将像下面这样来调用Jeeves:

```

% jeeves -s OO_Schema -t oo.tpl emp.om
Translated oo.tpl to oo.tpl.pl
Parsed emp.om
% ls *.h
Department.h Employee.h

```

这个模板现在能够按照你的规格说明为其中的每一个生成C++代码。在模板中一个小的改动就会立刻在所有的代码中反映出来。

## 噢，你也能这么做吗？

就当你完成了这项任务之后并准备回家时，你的非常有先见之明的老板走了进来，并让你再产生一个文件：一个用于创建相应关系数据库模式的SQL脚本。该脚本 *db.sql* 将会如下所示：

```
create table Employee (  
    id      integer,  
    name    varchar,  
    dept_id integer,  
)  
create table Department (  
    id      integer,  
    name    varchar,  
)
```

幸运的是，Jeeves 模板将使这项工作成为一个两分钟的活儿。你只需再创建一个模板文件（或者将例 17-2 中的内容添加到前面的模板中去）。

例 17-2: *sql.tpl*: 用于创建一个关系数据库模式的模板文件

```
@openfile db.sql  
@perl %db_typemap = ("int" => 'integer', string => 'varchar');  
@foreach class_list  
create table $class_name (  
    @foreach attr_list  
        @perl my $db_type = $db_typemap{$attr_type};  
        $attr_name $db_type,  
    @end  
)  
@end .. class_list
```

通过使用一段 Perl 代码，这个模板将每种属性类型映射到相应的 SQL 数据类型。

如你所看到的，这种体系结构允许我们重用规格语法分析器；我们通过使用由语法分析器产生的输出实现了一种完全不同的输出结果。

## Jeeves 概述

图 17-2 描述了一个基于 Jeeves 的翻译器的各种部件之间的关系。所有灰色的矩形构成了 Jeeves 框架。

Jeeves 框架提供了一个驱动程序，`jeeves`，一个模板分析模块，`TemplateParser.pm`，和一个用于创建与存取语法树的工具模块，`AST.pm`。

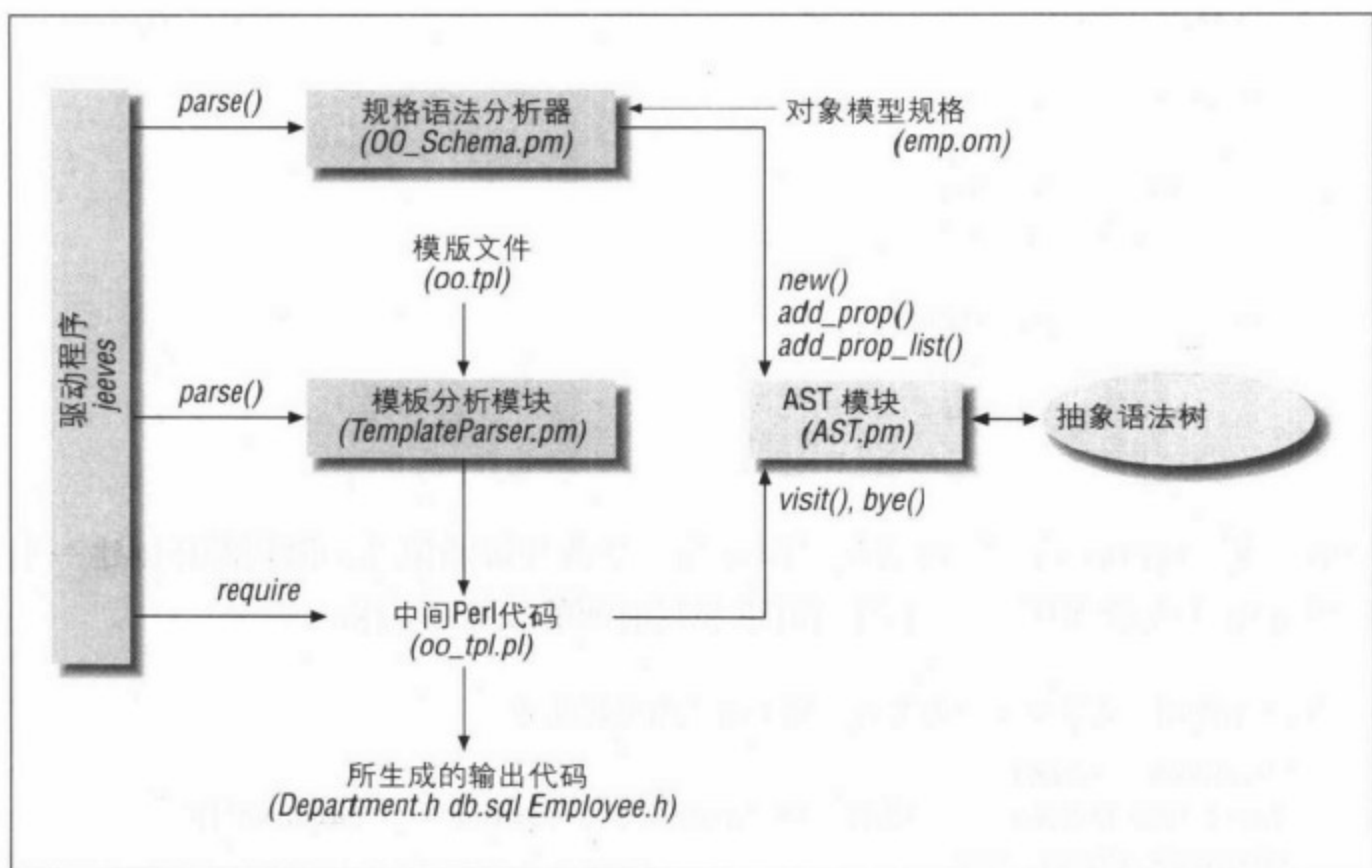


图 17-2 一个运行中的基于 Jeeves 的翻译器的组成部件

你提供一个给定应用领域的规格语法分析器，如 `OO_Schema.pm`，一个规格说明文件（`emp.om`）以及一个或多个模板文件，如 `oo.tpl`。

驱动程序首先将会调用规格分析器的 `parse` 函数。而它又调用 `AST` 的函数 `new`，`add_prop` 和 `add_prop_list` 来将所有相关的数据从规格文件转换成一个属性树。

驱动程序然后调用模板语法分析器的 `parse` 函数，由它将给定的模板文件转换成一个中间 Perl 文件（请注意我们前面提到的将 `oo.tpl` 翻译成 `oo.tpl.pl` 的命令行形



式), 模板中包含有变量及循环与条件结构, 所有这些都超出了 Perl 自身的支持能力, 因此通过把该模板转换成 Perl 代码, 我们可以更好的利用 Perl 的所有强大功能。这与早期版本的 C++ 编译器类似, 它们也是简单的将 C++ 文件转换成中间的 C 文件, 从而利用现有 C 编译器的功能, 优化特性和可移植性。

最后驱动程序通过 `require` 来加载中间文件, 而该中间文件其实就是伪装了的 `eval` 代码。当处理这个文件时, 其中的代码就会遍历语法树并产生所要求的输出文件。

## 这种体系结构的优势

那么我们从这种看起来复杂的组织形式得到了什么好处呢? 我们只编写一次输入语法分析器来标准化的数据结构。而模板语法分析器知道如何将这种数据结构应用在模板中, 并以一种可控制的方式遍历该数据结构。结果就是你可以编写各种各样的模板来重用输入语法分析器。

这种安排在大型的项目中工作得非常好。有语法分析经验的人可以编写语法分析器, 而对应用了解清楚的则来编写模板。其他的人只需编写不同的规格说明并运行这个工具。将来, 如果作为应用开发人员的你编写一个模板来自动从一种对象模型来产生基于 Motif 的用户界面, 那么其他人不必学习 Motif, 就可以为他们的对象模型产生定制的用户界面了。

Jeeves 驱动程序在命令行上要接受规格语法分析器的名字。这就是说, 你能够拥有一个针对各种类型问题的规格语法分析器库以及与这些语法分析器相对应的模板库。框件本身是独立于应用领域的。

我们使用 Perl 来编写它的好处就是, 没有别的语言能够与 Perl 的文本处理能力相匹敌。况且你还能够在你的模板中使用诸如 Tk 和 `IO::Socket` 等模块。

## 中间形式的 Perl 代码

Jeeves 的大部分代码都非常的简单; 唯一一处需要注意的代码就是模板语法分析器。

下面的代码段描述了一个从模板转换成中间形式的 Perl 文件的例子：

```
@foreach class_list
Name: $class_name
@foreach attr_list
Attr: $attr_name, $attr_type
@end
@end
```

我们去掉多余的只保留核心部分，相应的中间文件如下所示：

```
$ROOT->visit();
foreach $class_list_i (@class_list) {
    $class_list_i->visit();
    print "Name : $class_name\n";
    foreach $attr_list_i (@attr_list) {
        $attr_list_i->visit();
        print "Attr: $attr_name, $attr_type\n";
        Ast->bye();
    }
    Ast->bye();
}
```

Ast::visit将所有访问到的AST节点的属性转换成包main中的全局Perl变量。语法树的根节点将被首先访问到，这将会创建一个全局变量@class\_list。由于它就是根节点的首要的属性，而且当它们其中的一个被访问时，属性class\_name与attr\_list就会成为\$class\_name和@attr\_list以供使用。这段代码必须考虑到给定全局变量已经存在的可能性，这一般是由于嵌套外层有相似命名的属性，或者就是因为已经由模板通过一个@perl指令定义过了。因此visit()将会在需要时跟踪变量原先的值；bye将会在一个@FOREACH代码块结束时将其恢复成原先的值。

## Jeeves 的实现

在下面的章节中，我们将会实现Jeeves框架的所有部件。你会发现针对一个样例问题运行Jeeves，并将其输出的拷贝放在手边，会很有帮助。

## AST 模块

AST模块是一种非常简单的功能库,因此我们将在下面只了解其中几个比较有趣的过程。

一个AST节点就是一个属性包容器,因此散列表能够很好的胜任这项工作。为了调试方便,为每一个节点取一个名字:

```
package Ast;
use strict;
sub new {
    my ($pkg, $name) = @_;
    bless {
        'ast_node_name' => $name}, $pkg;
}
```

所有的规格语法分析器都使用new、add\_prop和add\_prop\_list来创建AST对象:

```
sub add_prop {
    my ($node, $prop_name, $prop_value) = @_;
    $node->{$prop_name} = $prop_value;
}
sub add_prop_list {
    my ($node, $prop_name, $node_ref) = @_;
    if (! exists $node->{$prop_name}) {
        $node->{$prop_name} = [];
    }
    push (@{$node->{$prop_name}}, $node_ref);
}
```

add\_prop只是简单的向AST对象增加一个名字-值对。add\_prop\_list创建一个列表值的属性。属性值为一个匿名数组,其中包含了指向其他AST节点的引用。你可以拥有自己的列表值属性,但是你决不能将它用做@foreach的参数,因为它所假定的列表元素为AST节点。

```
my @saved_values_stack;
sub visit {
    no strict 'refs';
    my $node = shift;
```

```

package main;
my ($var, $val, $old_val, %saved_values);
while (($var, $val) = each %{$node}) {
    if (defined ($old_val = $$var)) {
        $saved_values{$var} = $old_val;
    }
    $$var = $val;
}
push (@saved_values_stack, \%saved_values);
}

```

visit与bye方法由中间Perl文件来使用。\$node为正在访问的节点,而%\$node就是相应的散列表。\$var为诸如class\_name的属性名,因此为了检查\$class\_name这样的变量是否存在,我们可以使用符号引用,如if defined(\$\$var)。所有在此之前已经存在的变量,都被暂存在一个散列表(%saved\_values)中,而接着又被推送到一个堆栈中。这个栈就代表了所有这些暂存值的集合。

```

sub bye {
    my $rh_saved_values = pop(@saved_values_stack);
    no strict 'refs';
    package main;
    my ($var, $val);
    while (($var, $val) = each %{$rh_saved_values}) {
        $$var = $val;
    }
}

```

bye()简单的对栈进行弹出式操作,并将全局变量恢复到原先的值。附带说一句,由于use strict并不鼓励使用符号引用,因此我们不得不在用no stricts 'refs'时暂时将它关掉一会儿。

## 模板语法分析器

模板语法分析器支持表 17-1 中所示的一些指令。

表 17-1 Jeeves 所识别的指令

指令	描述
@//	注释。这一行不会被输出。
@foreach var [condition] @end	对 var 的每个元素进行循环（这里假定 var 为一个数组）并在条件（可选）成立时执行循环体。条件只是一些嵌入的 Perl 代码并可以这么来使用： @FOREACH attr_list (\$className eq "Test")
@if @elsif @else @end	直接翻译成 Perl 的 if 语句。（译注 2）
@openfile filename [options]	跟在这一行后面的所有语句将被简单的发送到这个文件中，直到碰到另一个 @OPENFILE 为止。其中的选项为： -append: 将文件以添加模式打开 -no_overwrite: 当文件存在时不要覆盖它 -only_if_different: 只在文件发生改变时才覆盖这个文件。这在 make 环境中非常有用，在那里你一般不想改变文件。
@perl	用于嵌入 Perl 代码，作为高级功能的换码符。 @perl \$user_name = \$ENV{USER}; @perl print \$user_name;

下面的模板语法分析器简单的将所有模板指令在中间文件中翻译成相应的 Perl 代码。每个子例程后面都有解释。

```

package TemplateParser;
use strict;

sub parse {
    # 参数：模板文件，中间 Perl 代码文件
    my ($pkg,$template_file, $inter_file) = @_;
    unless (open (T, $template_file)) {
        warn "$template_file : $@";
        return 1;
    }
    open (I, "> $inter_file") ||
        die "Error opening intermediate file $inter_file : $@";

```

译注 2：原文如此。实际代码中并未实现。

```

emit_opening_stmts($template_file);
my $line;
while (defined($line = <T>)) {
    if ($line !~ /^s*\@/) { # Is it a command?
        emit_text($line);
        next;
    }
    if ($line =~ /^s*\@OPENFILE\s*(.*)\s*/i) {
        emit_open_file ($1);
    } elsif ($line =~ /^s*\@FOREACH\s*(\w*)\s*(.*)\s*/i) {
        emit_loop_begin ($1,$2);
    } elsif ($line =~ /^s*\@END/i) {
        emit_loop_end();
    } elsif ($line =~ /^s*\@PERL(.*)/i) {
        emit_perl("$1\n");
    };
}
emit_closing_stmts();

close(I);
return 0;
}

```

TemplateParser::parse 由驱动程序以模板文件名为参数进行调用。对模板中的每一行，它都要检查该行是一条命令还是普通文本，并相应的调用子例程“emit”。所有产生的代码以斜体显示。

```

sub emit_opening_stmts {
    my $template_file = shift;
    emit("#Created automatically from $template_file");
    emit(<<'_EOC_');
    use Ast;
    use JeevesUtil;

    $tmp_file = "jeeves.tmp";
    sub open_file;
    if (! (defined ($ROOT) && $ROOT)) {
        die "ROOT not defined \n";
    }

    $file = "> -"; # 假定为 STDOUT, 除非 @OPENFILE 改变了它.
    open (F, $file) || die $@;
    $code = "";

```

```

$ROOT->visit();
_EOC_
}

```

所有写入（发送）到中间文件的代码都以斜体形式显示。在这里大量使用了 Perl 的“原地文档”特性，因为这样我们可以不受限制的使用引号和换行符。emit\_opening\_statement 用来访问语法树的根节点（驱动程序以一个称做 \$ROOT 全局变量形式提供对根节点的访问）。默认情况下，所有从中间文件的输出都被送往标准输出，直到碰到一个 @openfile 指令为止。

```

sub emit_open_file {
    my $file = shift;
    my $no_overwrite = ($file =~ s/-no_overwrite//gi) ? 1 : 0;
    my $append = ($file =~ s/-append//gi) ? 1 : 0;
    my $only_if_different = ($file =~ s/-only_if_different//gi) ? 1 : 0;
    $file =~ s/\s*//g;
    emit (<<"_EOC_");
    # 行$
    open_file("\$file", $no_overwrite, $only_if_different, $append);
    _EOC_
}

```

emit\_open\_file 中包含了对 @openfile 的翻译处理并简单的产生一个对工具函数 open\_file（我们将在后面讨论）的调用。

```

sub emit_loop_begin {
    my $l_name = shift; # 数组变量名
    my $condition = shift;
    my $l_name_i = $l_name . "_i";
    emit (<<"_EOC_");
    # 行$
    foreach \$$l_name_i (\@{$l_name}) {
        \$$l_name_i->visit ();
    _EOC_
        if ($condition) {
            emit ("next if (! ($condition));\n");
        }
    }
    sub emit_loop_end {
        emit (<<"_EOC_");
    }
    # 行$
    Ast->bye();
}

```



```

}
_EOC_
}

```

我们前面已经见到了根据@foreach指令产生的代码。请注意我们怎样生成迭代器的名字，以及如何保护特定的表达式免于替换操作。如果看以下样例的输出结果，就能更好的理解这段代码。

```

sub emit {
    print I $_[0];
}
sub emit_perl {
    emit($_[0]);
}
sub emit_text {
    my $text = $_[0];
    chomp $text;
    # 对文本中的引号进行
    $text =~ s/"\\/"/g;
    $text =~ s/\\ /g;
    emit("<<\"_EOC_\"");
output("$text\\n");
_EOC_
}

sub emit_closing_stmts {
    emit("<<\"_EOC_\"");
Ast::bye();
close(F);
unlink ($tmp_file);
sub open_file {
    my ($a_file, $a_nooverwrite, $a_only_if_different, $a_append) = @_;

    # 首先处理前面打开的文件
    close (F);
    if ($only_if_different) {
        if (JeevesUtil::compare ($orig_file, $curr_file) != 0) {
            rename ($curr_file, $orig_file) ||
                die "Error renaming $curr_file to $orig_file";
        }
    }
    # 现在是新文件
    $curr_file = $orig_file = $a_file;

```



```

$only_if_different = ($a_only_if_different && (-f $curr_file))
                    ? 1 : 0;
$no_overwrite = ($a_nooverwrite && (-f $curr_file)) ? 1 : 0;
$mode = ($a_append) ? ">>" : ">";
if ($only_if_different) {
    unlink ($tmp_file);
    $curr_file = $tmp_file;
}
if (! $no_overwrite) {
    open (F, "$mode $curr_file") || die "could not open $curr_file";
}
}

sub output {
    print F @_ (! $no_overwrite);
}
1;
_EOC_
}

```

`open_file`与`output`子例程出现在所有中间代码文件中(并没有什么特别的原因——它们还不如被放在JeevesUtil包中)。`open_file`将关闭前面打开的文件。如果你这么来写, `@openfile foo -only_if_different`, 那么中间文件将会把模板输出写入到一个临时文件中, 并在完成后将这个临时文件与 `foo` 的内容进行比较, 然后只有在二者存在差异时才覆盖这个文件。

## Jeeves 驱动程序

*Jeeves*脚本只不过是个驱动程序, 它首先调用模板分析器来产生中间文件, 然后调用输入分析器(实际上是它的`parse`方法)产生语法树, 最后对中间文件进行计算(`eval`)。

模板文件只有在比中间文件更新时才被重新编译。

例17-3给出了*Jeeves*的代码, 省略了没什么意义的部分(如`process_args()`)。

例17-3: Jeeves

```

#!/opt/bin/perl
# process_args 初始化下面的全局变量

```

```

# $spec_file      - 输入规格文件名(emp.om)
# $template_file  - 模板文件名(oo.tpl)
# $inter_file     - 中间文件名
#                  (默认为$template_file.pl)
process_args();
#-----
# 对模板文件进行语法分析
#-----
# 使用 "require" 来让 process_args() 首先设置 @INC
require 'TemplateParser.pm';
my $compile_template = 1;
if ((-e $inter_file) &&
    (-M $inter_file) >= (-M $template_file)) {
    $compile_template = 0; # 如果中间文件更新则不要进行编译
}
if ($compile_template) {
    if (TemplateParser->parse ($template_file, $inter_file) == 0) {
        print STDERR ("Translated $template_file to $inter_file\n")
            if $verbose;
    } else {
        die "Could not parse template file - exiting\n";
    }
}
#-----
# 对输入规格文件进行语法分析
#-----
require "${spec_parser}.pm"; $spec_parser->import;
$ROOT = $spec_parser->parse($spec_file);
print STDERR ("Parsed $spec_file\n") if $verbose;
$ROOT->print() if $debugging;
#-----
# eval 中间 Perl 代码文件
#-----
require "$inter_file";
die "$@ \n" if $@;
exit(0);

#-----
sub Usage {
    print STDERR <<"_EOT_";

Usage: jeeves <options> <specification file>
where options are:
-t <template file>           : Name of the template file.

```

```

Default : "../jeeves.template"
Default template directory = ".", which
can be modified by setenv-ing
"JEEVESTEMPLATEDIR"

-q           : Quiet Mode
-d           : Set a debugging trace. This is NOT quiet!
-s <specification parser> : Parser module that can parse the input
                        specification file
                        Default : "oo_schema"
[-ti <intermediate perl file>] : jeeves translates the template file to
                        : perl code. Default : "<template>.pl"
-D var[=value] : Define variables on the command line

```

The command line can be specified in the envt. variable "JEEVESOPTIONS".

```

The pathname to all Jeeves modules can be set in the envt. variable
  "JEEVESLIBDIR" (colon-separated);
__EOT__
  exit(1);
}

```

## 规格语法分析器样例

输入规格语法分析器特定于某一应用领域。这一节将来看一下我们的业余对象模型规格分析器，我们的主要目的就是复习一下AST库是如何使用的；语法分析器代码本身非常简单。对于更复杂的语法分析任务，你可以使用 Berkeley yacc 的一个版本，该程序经过改造能够产生 Perl 而不是 C 代码（获取地址为 <http://ftp.sterling.com:/local/perl-byacc.tar.Z>）。我已经应用这种组合成功的生成了 CORBA 规范的 IDL 语法分析器。

例 17-4 中的语法分析器允许属性具有如下的附加注语：

```

class Foo {
    int id, access=readonly, db_col_name=id, index=yes;
};

```

在模板文件中这些标志属性可以像“标准”属性如 *attr\_name* 和 *attr\_type* 一样来使用。

```

package SchemaParser;
use Ast;
use Carp;
sub parse{
    my ($package, $filename) = @_ ;
    open (P, $filename) || die "Could not open $filename : $@";
    my $root = Ast->new("Root");
    eval {
        while (1) {
            get_line();
            next unless ($line =~ /^\\s*class +(\w+)/);
            $c = Ast->new($1);
            $c->add_prop("class_name" => $1);
            $root->add_prop_list("class_list", $c);
            while (1) {
                get_line();
                last if $line =~ /^\\s*}/;
                if ($line =~ s/^\\s*(\\w+)\\s*(\\w+)/) {
                    $a = Ast->new($2); # 属性名
                    $a->add_prop("attr_name", $2); # 属性类型
                    $a->add_prop("attr_type", $1); # 属性类型
                    $c->add_prop_list("attr_list", $a);
                }
                $curr_line = $line;
                while ($curr_line !~ /;/) {
                    get_line();
                    $curr_line .= $line;
                }
                @props = split (/[,;]/, $curr_line);
                foreach $prop (@props) {
                    if ($prop =~ /\\s*(\\w*)\\s*=\\s*(.*)\\s*/) {
                        $a->add_prop($1, $2);
                    }
                }
            }
        }
    };
    # 如果抛出了例外 "END OF FILE" 就会运行到这里
    die $@ if ($@ && ($@ !~ /END OF FILE/));
    return $root;
}
sub get_line {
    while (defined($line = <P>)) {
        chomp $line;
    }
}

```

```
    $line =~ s#//.*$##;          # 删除注释
    return if $line !~ /^\\s*$/;  # 如果不是空白符就返回
}
die "END OF FILE";
}
1;
```

OO\_Schema::parse 开始先创建一个新的 AST 根节点，并在每遇到一个新的类声明时，将其加入到根的 *class\_list* 属性中。与之相似，它为每一个属性创建一个新的节点并将其加入到代表当前类的 AST 节点的属性 *attr\_list* 中。

当没有东西可读时，过程 *get\_line* 将会抛出一个文件结束例外。这样的话，*get\_line* 的用户就能够将多条对 *get\_line* 的调用放在一个 *eval* 包裹中，而不必在每个地方都检查是否意外的到达了输入的结尾。

## 相关资源

1. Berkeley yacc for Perl。

可以从 CPAN 的 *src/misc/perl-byacc.tar.Z* 下获得。

2. Lex for Perl. *Parse::Lex*。

可以从 CPAN 获得，使用 Lex 来生成词法分析器。文档是以法语写成的，即便是你不会说法语，那些功能库也很容易看懂。

3. Research Issues with Application Generators, Proceedings of the 6th Annual Workshop on Software Reuse. Prem Devanbu。

4. A Configurable Code Generator for OO Methodologies, A. Aimar, A. Khodabandeh, P. Palazzi 和 B. Rousseau。

地址为 [http://www1.cern.ch/WebMaker/examples/CHEP94\\_codegene\\_1/www/codegene\\_1.html](http://www1.cern.ch/WebMaker/examples/CHEP94_codegene_1/www/codegene_1.html)。

5. Little Languages, More Programming Pearls: Confessions of a Coder. John Bentley. Association for Computing Machinery, 1988。

6. Building Application Generators, J. Craig Cleaveland. IEEE Software, July 1988。
7. Tools for Building Application Generators, J. Craig Cleaveland 和 Chandra M.R. Kintala. AT&T Technical Journal, July/August 1988。
8. Thank You, Jeeves. P.G. Wodehouse. Aeonian Press, 1983。



本章简介:

- 编写一个扩展: 概述
- 例子: Perl 与分形计算
- SWIG 的功能
- XS 的功能
- 自由度
- 分形介绍

# 第十八章

## 扩展 Perl:

### 第一课

第一个望远镜制造者的汤普森规则:

“先做一个四英寸的镜片, 接着再做六英寸的镜片,  
要比直接去做一个六英寸的镜片来的快。”

——Jon Bentley *《Programming Pearls》*

与使用系统编程语言相比, 脚本语言差不多总是一种更加令人愉快和多产的另类解决方案。然而脚本语言并不是天生就能做任何事情的(注1), 你有时候仍然需要深入 C/C++ 来获得速度优势, 精细的数据结构, 类型安全以及对现有功能库的存取。与 *awk* 和早期版本的 BASIC 相比, Perl, Visual Basic, Python 和 Tcl 这些语言因为具有很好的与 C 语言结合的能力, 获得了严肃开发语言的地位, 而前者则很少用于产品化应用系统的开发。

在这一章, 我们来查看一下将 Perl 与 C 代码粘合在一起的是些什么东西, 然后再来学习两个帮助我们完成大量此类联编任务的工具集。其中第一个是一对称做 *h2xs* 和 *xsubpp* 的工具, 它们与 Perl 发行版捆绑在一起。我们将这对工具简称为 XS(注2), 因为它还涉及了一种与之具有相同名字的中间语言。另一种工具就是由犹他大学的 Dave Beazley 编写的 SWIG (Simplified Wrapper and Interface Generator, 简化包裹程序和接口生成程序)。

注1: 就 Perl 而言, 搞清“任何事情”的定义有点儿难。

注2: XSUB 与 XS 均代表 eXternal SUBroutine (外部子程序)。

我们讲述这些工具的一些常用的功能子集，并且将会了解到即使是对内部 Perl API 一无所知也能完成大量的工作。但是对于许多强大的功能，则要等到我们学到第二十章“Perl 的内部工作”中的“丰富的扩展”一节才能实现。

本章要求你已经准备好了下面的这些模块：C::Scan，Data::Flow。这两个模块均由 *h2xs* 使用，它们可以从 CPAN 获得。还有就是用于创建 GIF 文件的 *gd* 库，它可以从 [www.boutell.com](http://www.boutell.com) 处下载。

## 编写一个扩展：概述

图 18-1 描述了一个名为 *testmatrix.pl* 的文件，它底层调用了用一个用 C 编写的矩阵库。为了将这两组代码编联到一起，我们需要一些粘连代码（glue code），如黑灰色方框所示。

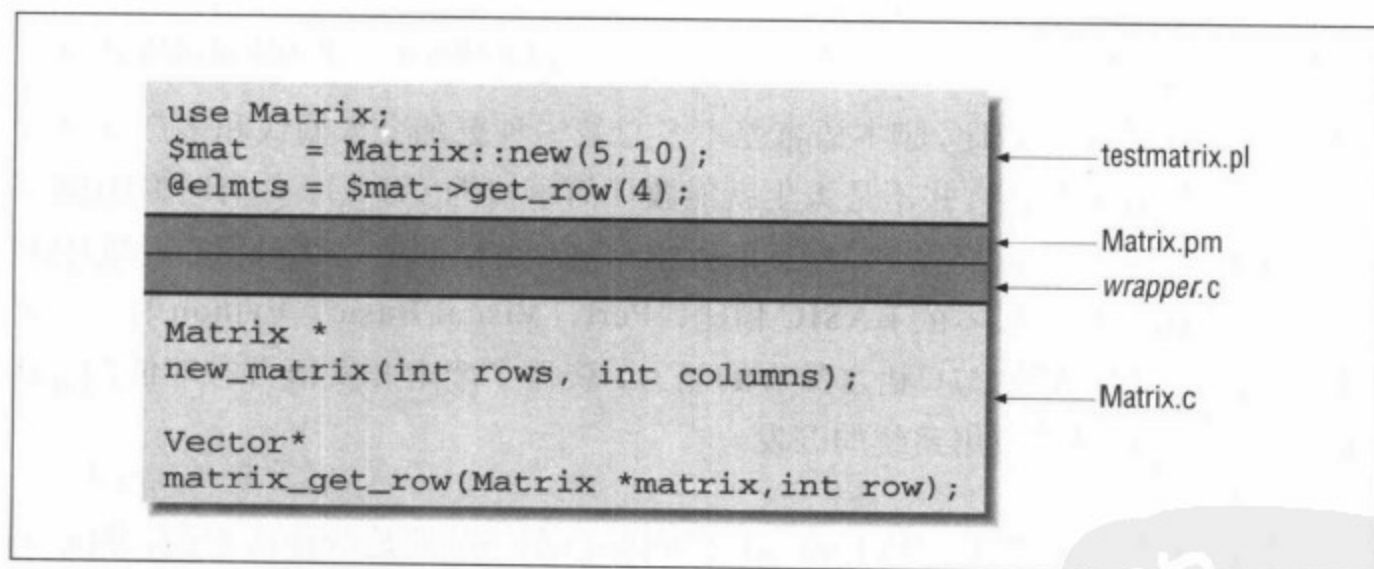


图 18-1 从 Perl 中调用 C

XS 和 SWIG 均将创建的粘连代码放置到两个文件中——一个 Perl 模块文件与一个 C 包裹文件。而且我们还需要处理以下问题：

### 数据类型转换

Perl 的标量变量参数可以方便的转换成诸如 *int*，*double* 或 *char \** 的基础 C 数据类型（反之也一样）。处理用户定义的数据结构如 *Matrix \** 或 *Vector \**



等则要棘手的多。图 18-1 中的 `$mat` 包含了一个指向用户定义数据类型的 C 指针。`xsubpp` 与 SWIG 都提供了类型映射机制。这种机制允许你编写定制的代码来处理 Perl 与非一般的 C 数据结构的之间的转换。你必须了解一些内部的 API 才能编写类型映射, 因此我们将在第二十章再来讨论这个问题。

### 内存管理

Perl 自动的管理为用户定义变量分配的内存空间, 而 C 则要求程序员来处理所有的细节。当数据跨越 Perl 与 C 的接口时显得尤为重要。不幸的是, C 函数从外表看不能提供任何有关其内存管理的方式的线索; 开发人员都很难推测出来, 更不要说像 SWIG 和 XS 这样的自动工具了。让我们假定 C 矩阵库在内部是以一系列矢量对象来存储数据的 (每一行都表示为一个矢量), 而 `matrix_get_row` 则返回与某一行对应的矢量对象。如你所见, `new_matrix` 与 `maxtri_get_row` 均返回指向对象的指针。但是对于前者, 调用者需要拥有对象的所有权 (当不再需要它时就将其删除), 而对于后者则由矩阵库拥有所分配的存储空间。虽然扩展工具提供了某些默认的选择, 然而你必须时刻保持警惕。你还要确保使用恰当的函数来释放 (`free`, `delete`, `delete[]`) 分别由 `malloc` 或 C++ 的 `new` 或 `new[]` 分配的存储空间。

### 方便在 Perl 中的使用

像:

```
($a,$b,$c) = $mat->get_rwo(10);
```

这样一个简单的调用都涉及到众多 Perl 的特性, 如包、不定数量的函数参数、从函数的多值返回、面向对象的记号、`wantarray` 功能等等。一个扩展模块应当尽可能的让 Perl 程序员感受到 Perl 的风格。

### 引导与初始化

对于要在 Perl 中调用的 C 库来说, 它需要被静态的或动态的连接到 Perl 解释器中。由 XS 和 SWIG 所产生的 Perl 模块就包含了用于进行引导和初始化 C 库的代码。(上面所描述的其他函数均出现在 C 包裹代码中。)

## 扩展流程

C 头文件 (如 `Matrix.h`) 中包含了数据结构声明、预处理宏、公共存取的变量以及函数原型——实质上就是 C 库的接口。你通常不会允许从 Perl 脚本中存取所

有的功能；没有任何比在 Perl 中尝试 C 编程更糟糕的了。在大多数情况下，开放一些公共函数的子集以及一些常量（一般以初始化过的变量，`#define`或`enum`的形式出现）就足够了。我们统一将它们称之为公共接口，并将它们抽取到一个公共头文件中。

图 18-2 描述了 Matrix 库的头文件是如何被用在这两种工具集的输入的。

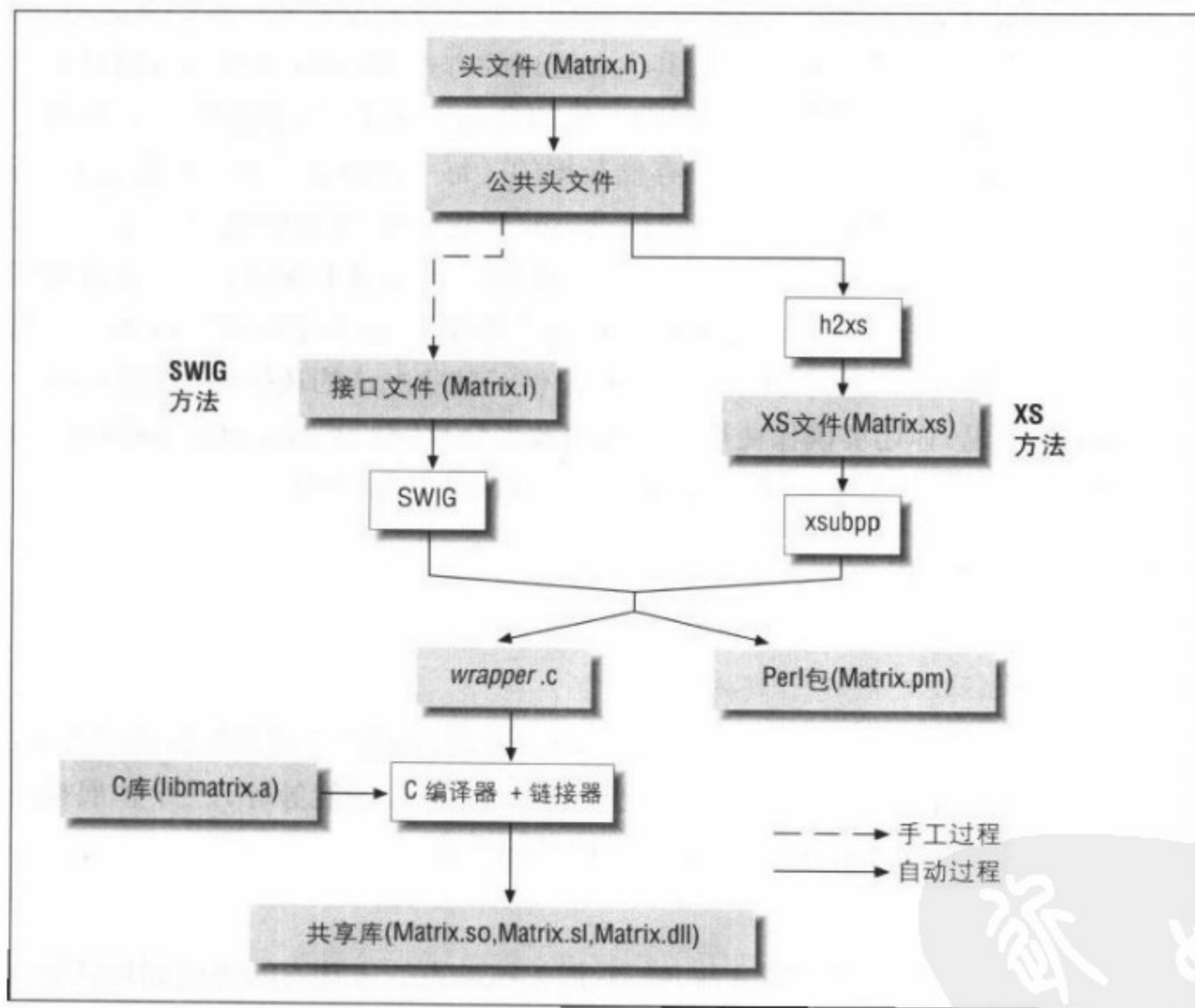


图 18-2 SWIG 和 XS 流程

公共头文件中可能包含复杂的 C 声明。SWIG 需要作为扩展开发人员的你将其精简为比较简单的形式，并以接口定义语言的形式表达出来。幸运的是，这种语言与 ANSI C 和简单的 C++ 如此接近，以至于大量的头文件都无须任何转换工作要做。SWIG 将会从接口描述产生粘连代码；就 Matrix 的情况而言，产生的就是

*Matrix.pm* 和 *Matrix\_wrap.c*。如果你的系统支持动态连接（在 Unix 上就是共享库，在 Windows 上就是 DLL），而且如果 Perl 可执行程序建立时支持使用它的话，那么所有要做的工作就是将粘连代码和你的 C 库转换成一个动态连接库。如果没有动态连接这个选项，那么将会通过静态的连接 Perl 功能集库（Unix 上是指 *libperl.a*，Windows 上则是 *perl.lib*）与上面所提到的代码，来生成一个新的 Perl 可执行文件。

*h2xs* 与 *xsubpp* 采用的方式稍有不同。*h2xs* 理解 C 头文件（但不是 C++），并将所有的常量和函数原型都转换成一种称做 XS 的元语言。但是对于脚本编程来说，函数声明或许仍然太复杂了一些，因此这种方式需要你摆弄一下 *h2xs* 所产生的 *.xs* 文件，并采取必要的措施来简化接口。当然如果接口已经足够简单，那么就无须进行手工的转换工作。XS 语言是一种 C 与一些奇特关键词的混合体，并提供了让你重写由 *xsubpp* 所产生的粘连代码的指令。

巧的是，由这两种工具所产生的代码非常相似，而且完全可以使用 XS 的方式来建立一些扩展，而使用 SWIG 来创建另一些扩展。那么我们会问，应当使用哪一种呢？

## SWIG 还是 XS？

SWIG 与 XS 在设计目标上的不同导致了它们具有不同的特性。SWIG 被设计用来帮助创建位于 C 上的脚本包裹层，它不但支持 Perl 还支持 Python、Tcl 和 Guile。相比之下，XS 只为 Perl 而设计，而且还支持许多 Perl 所特有的模式，而这些对于 SWIG 来说是无法轻而易举在其他语言中实现的。

我更倾向于使用 SWIG，因为它要简洁的多，不像 XS 那样要涉及很多内部问题，而且还支持多种语言。此外它还拥有对数据结构优异的支持（而不仅仅是函数），而 XS 却只提供对函数的支持。我的职业就是创建 C++ 和 Java 应用，因此我通常关注更多的是应用而不是前端的脚本编程——我让用户来选择脚本语言。你的经历或许不同。

你将会发现当前 Perl 发行版中以及 CPAN 中的所有模块，均是以 XS 来编写的。主要原因就是 XS 与 Perl 捆绑发行。况且它还在一开始就支持诸如类型映射等强

大功能，而SWIG是在近期才提出来的。如果你需要理解或修改任何CPAN中的模块，你则必须要了解XS。

这两种工具都提供了相当的自由度来弥补其大多数的缺点，因此我建议随便挑一个只管去用。

## 例子：Perl 与分形计算

总体上我们已经讨论得够多的了，现在让我们通过一小段绘制分形图的代码来试用一下这两种工具。这种问题是为C量身定做的，因为产生一幅分形图需要涉及对每个点的一系列计算，这要求有紧凑的数据结构和快速的数字计算。这个练习将产生我们所熟悉的 Mandelbrot 集分形图，如图 18-3 所示。

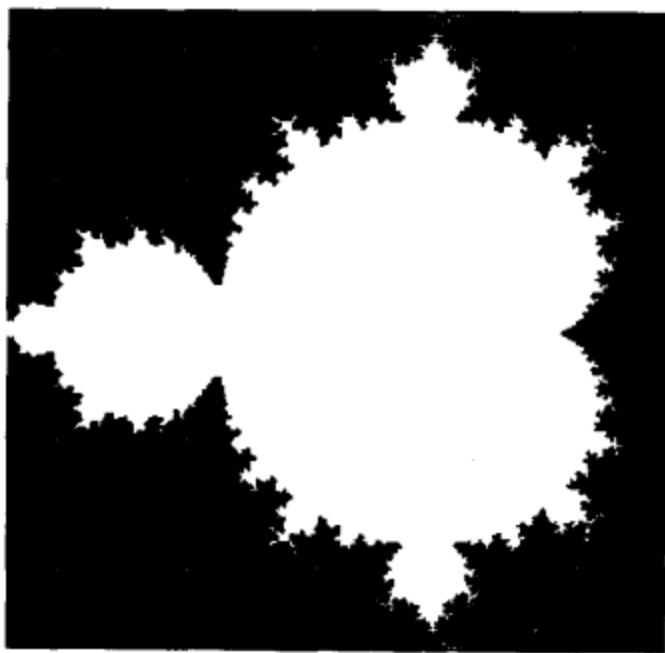


图 18-3 Mandelbrot 集

我们的 Mandelbrot 代码在文件 *mandel.c* 和 *mandel.h* 中实现。为了避免不可移植的 GUI 解决方案，我们使用了一个由 Tom Boutell 编写的公共域库 *gd*，它可以允许你将一个 GIF 文件当作画板，并在上面绘制点、线和圆。之后，这个 GIF 文件则可以使用任何浏览器来观看。

*mandel.c* 中实现了一个名为 *draw\_mandel* 的函数，其原型如例 18-1 所示。

例 18-1: Mandel.h

```
extern int
draw_mandel (char *filename,
             int width, int height,
             double origin_real, double origin_imag,
             double range, double depth);
```

其中参数的意义将在本章后面的“分形”一节中进行解释。首先我们将把注意力放在如何使它可以在 Perl 中进行调用。

## 使用 SWIG 进行分形调用

我们先来编写一个 SWIG 接口文件，Fractal.i，如例 18-2 所示。

例 18-2: Fractal.i —— SWIG 接口文件

```
%module Fractal
%{
#include "mandel.h"
%}
#include mandel.h
```

%module 语句为该文件中的所有接口声明提供一个唯一的名字空间。由于我们想让所有的分形绘制代码都拥有一个名字空间，而Mandelbrot集只是其中的一个选择，因此我们称该模块为 Fractal。

{ 与 %} 之间的语句用以指代“原始”C 代码。我们之所以在这里包含 *mandel.h*，文件是因为该接口文件马上要被转换成 C 粘连代码，而转换后的代码需要这个头文件。现在到了列出所有数据结构和要输出的函数的（带有完整的声明原型）部分了。由于接口文件与 ANSI C 非常接近，我们只需简单的 *%include mandel.h*。与第一个包含不同，这个包含以 % 开始，因为它马上将在 SWIG 中被调用，而前者是以 # 开头，这是因为它要在以后的 C 代码中调用。

下一步我们以这个接口文件为参数调用 SWIG，并指定脚本语言为 Perl5:

```
% swig -perl5 Fractal.i
Generating wrappers for Perl 5
% ls
```

```

mandel.h      mandel.c      Fractal_wrap.doc
Fractal.i     Fractal.pm     Fractal_wrap.c

```

SWIG 根据接口文件创建了四个文件。*Fractal.pm* 中包含了使 C 库可动态加载的代码。*Fractal\_wrap.c* 中包含了包裹代码；对于接口文件中的函数 *foo*，其包裹文件将会包含一个名为 *\_wrap\_foo* 的函数，由它将 Perl 参数值转换为 C 语言的形式，并调用 *foo*，然后将返回结果以 Perl 数据类型打包返回。你无须理解文件 *Fractal.pm* 和 *Fractal\_wrap.c* 的内容。SWIG 还从接口文件中抽取所有文档并放入到 *Fractal\_wrap.doc* (ASCII)，或 *Fractal\_wrap.html* (HTML)，或 *Fractal\_wrap.tex* (LaTeX) 中。

我们剩下要做的就是编译这两个 *.c* 文件并使它们可动态加载（注 3）。同样 SWIG (XS 也是如此) 还帮助你创建一个 *makefile* 来简化这部分工作。因为 *makefile* 要依赖于机器和特定的环境如操作系统的特性、编译器、连接选项等等因素，因此这些工具并不是直接产生一个 *makefile*，而是生成一个称做 *Makefile.PL* 的 Perl 脚本程序，而当该程序执行时则会产生一个为你的系统所定制的 *makefile*。这个脚本非常简单，在我们手工加上了 LIBS 和 OBJECT 行之后的样子如下所示：

```

use ExtUtils::MakeMaker;
WriteMakefile(
    'NAME'    => 'Fractal',          # 模块名
    'LIBS'    => ['-L/usr/local/lib -lgd'], # 要连接的定制功能库
    'OBJECT'  => 'mandel.o Fractal_wrap.o' # 所有目标文件
);

```

标准的 ExtUtils::MakeMaker 模块完成了查找你的系统配置和创建定制 *makefile* 的奇妙工作。

下面的三步建立并安装了这个扩展模块：

```

% perl Makefile.PL      # 创建 Makefile
% make                  # 编译源文件并创建共享库
% make install           # 可选择的功能库安装

```

注 3： 由于这是一种比静态连接简单得多的选项，而且大多数够格的操作系统都支持它，所以本书只考虑动态连接。

(你还需要比这个更容易更具可移植性的方式吗?)

我们现在完全做好了创建分形图的准备工作。下面的对 `draw_mandel()` 的调用创建了图 18-3 中所示的美丽的画面。

```
use Fractal;
Fractal::draw_mandel('mandel.gif', 300, 300,      # file, width, height
                    -1.5, 1.0,                    # 起始 x, y
                    2.0, 20);                     # 范围, 最大迭代数
```

由于本章的主要目的是描述如何编写扩展模块, 因此我们将(不情愿的)对 `draw_mandel` 的讨论放到最后来讲。

## 使用 XS 进行分形调用

XS 过程也是直接了当的。由于 `h2xs` 理解普通的 C 头文件, 因此分形扩展将由下面的语句产生:

```
% h2xs -x -n Fractal mandel.h
```

它将会产生 Perl 模块 `Fractal.pm`, makefile-generating 脚本 `Makefile.PL` 和 `Fractal.xs`。现在不必知道该文件中都包含了什么内容。

由于 `Makefile.PL` 是自动生成的, 因此同前面一样, 你将会需要增加或修改 OBJECT 与 LIB 行。扩展的建立与安装方式与我们前面看到的完全一样:

```
% perl Makefile.PL
% make
% make install
```

前面一步生成的 makefile 将会注意到 `Fractal.xs` 的存在, 并把它提供给 `xsubpp` 来产生 `Fractal.c` 中的粘连代码。请注意一下文件名不是使用 SWIG 时的 `Fractal_wrap.c`, 因此 `Makefile.PL` 中的 OBJECT 一行应当是下面的这个样子:

```
'OBJECT' => 'mandel.o Fractal.o' # mandel.o 中包含有实际的函数
                        # Fractal.o 中包含有粘连代码
```

## SWIG 的功能

在完成了一趟编写扩展模块的旅行之后，让我们再来仔细看一看SWIG的众多功能。我们前面提到过SWIG可以处理一部分有用的ANSI C/C++子集，也就是说它不但支持函数还支持数据结构。具体的讲，它支持下面的内容：

### 常量与全局变量

一个C变量可以以一个拥有相同名字的标量变量的形式，输入到Perl空间中。SWIG支持基础C数据类型，enums和#define常量。具有复杂或用户定义数据类型的变量，将会自动的映射到一对get/set存取函数。

### 指针

默认情况下，每个指针都被当作void\*，而不管它是char\*\*还是Matrix\*或是double\*\*\*。这种策略对于用户定义的数据类型来说工作的非常好，这是因为大多数C库并不需要你来解析这些指针。例如，fopen返回一个FILE\*，然后简单的交给fread()和fwrite()。在Perl中以标量变量的形式进行使用，而Perl不必知道该指针指向的是一个数组，还是结构，或是一个typedef。另一方面，如果你需要一个指向具有整数值标量变量的列表的Vector\*的话，那么你就必须提供给SWIG一个类型映射。

### 类型映射

并不是所有的数据类型都是Perl与C之间的简单转换。SWIG（与xsubpp类似）为你提供了一种编写任意转换的方法，如把一个Perl数组转换成10x10的矩阵。要编写一个类型映射，你需要了解用于存取内部数据类型的Perl API。因此我们将在第二十章的“SWIG的类型映射”一节再来讨论这个话题。类型映射不仅可以应用于函数参数，还可以用在结构成员和全局变量中。你还可以选择创建有名类型映射，将其应用于特定的有名实体（函数参数，变量名，函数名），而不是那个类型的所有实体。

### 数组

无论是简单数组（vector[100]）还是多维数组（vector[10][10]）都被映射到一个简单的指针（vector\*）。存在支持数组的类型映射，但是仍然存在许多SWIG不能提供一致解决方案的棘手问题；请你查阅SWIG文档来获得更详细的信息。



### 结构与 C++ 类

SWIG 自动为每一个定义在接口文件中的结构成员或类创建存取函数。与其他机制一样, 这些声明并不能够完全的拥有 C 结构或 C++ 类的多数特性, 但是它们完全能够胜任处理大多数常见接口的问题。

### 方法

SWIG 提供构造函数与析构函数, 它们可以允许你从 Perl 空间中分配和释放 C 结构。你可以使用一个名为 `%addmethods` 的原语, 将基本的 C 结构转换成 Perl 空间中的对象。

### 普通函数

SWIG 可以创建与它们的 C 函数非常相似的包裹函数。每个参数都可以进行有选择的类型映射, 但是由于一个类型映射提供了一种与其他参数分离的转换方式, 因此参数个数不能改变。这在 XS 中就不受这个限制。

换句话说, 使用 SWIG 你不能够将 C 函数

```
char ** permute(char *string); // 返回字符串的排列
```

映射到

```
@array = permute ($str);
```

这是因为其中的一个参数 `char **` 需要转换成一组个数不定的标量变量(来赋值给 `@array`)。你可以编写一个将 `char **` 转换成一个数组的类型映射, 并返回它的引用, 因此在 Perl 空间中, 它可以这样来存取:

```
$rarray = permute ($str);  
print join(' ', @$rarray);
```

当然你还可以编写一个 Perl 包裹函数, 并将其插入到由 SWIG 自动生成的 `.pm` 文件中:

```
sub fancy_permute {  
    @{permute($_[0])}; # 数组引用间接访问  
}
```

### 默认及可选参数

同在 C++ 中一样, 参数可以有默认值, 但是只能应用在最右端的参数上。下面是你如何在接口文件指定函数的原型的例子:

```
draw_mandel (file,width,height,orig_real,orig_imag,range,depth=30);
```

它允许你在从 Perl 中调用时可以选择跳过最后一个参数。

### 集中化的例外处理

SWIG 提供一个 `%except` 指令, 将所有的外部库调用包裹在一个通用的例外处理函数中。这样你就可以集中捕获所有用户定义的错误和 C++ 例外, 并将它们转换成 Perl 例外。请阅读 SWIG 文档来查找相关的例子。

### 影子类 (*shadow class*)

SWIG 可以选择创建允许你使用 Perl 的散列记号, 来存取 C 或 C++ 对象的成员属性和函数的 Perl 包裹代码, 如 `$person->{age}`。这种机制是建立在前面提到的属性存取函数的基础上的。

### 嵌套结构

嵌套结构可以获得与最外层结构相同的待遇 —— 包括存取方法和创建影子类。

下面的接口文件描述了一个使用类, 存取方法和创建影子类的例子:

```
%module Graphics

class Shape {
public:
    int x, y; // 初始值
    int w, h; // 宽度, 高度 (定义限定区域)
    draw();
};

class Polygon : public Shape {
public:
    Polygon(int x, int y, int w, int h);
    draw();
};
```

由于默认状态下不是被打开的, 我们将使用 `-c++` 选项和用于创建影子类的 `-shadow` 选项来调用 SWIG:

```
% swig -c++ -shadow Graphics.i
```

SWIG 在脚本中建立了相同的继承层次并在 Perl 中使用这个类, 这让人感觉非常自然:

```
use Graphics;
$poly = new Polygon(10, 10, 30, 40);
printf "Origin: %d %d \n", $poly->{x}, $poly->{y};
$poly->draw();
```

你会很高兴的知道 SWIG 可以恰当的处理基类与导出类之间的关系。比如, 一个包含基类的函数将会识别 `bless` 到导出类中的指针。对于多重继承的情况, SWIG 将会恰当的进行类型转换, 以确保指针类型的正确性。XS 中没有这种功能。

虽然影子类的功能很方便, 但是你应当意识到对于每一个使用 `new` 来产生的实例在内部都还要创建一个附加的对象。原因就是支持成员存取记号 (`$poly->{x}`), `new` 返回了一个绑定的散列表, 而它的 `FETCH` 子例程将会调用相应的存取函数。现在你就会知道是绑定机制安插在中间对象里面了。

## XS 的功能

我们前面提到过, XS 是一种接口定义语言。与 SWIG 不同, XS 主要专注于 C 函数和 `#define` 常量, 而不提供任何对 `struct` 或 `class` 定义的支持 (尽管有计划在将来要做到这一点)。在实际应用中, 我并不十分怀念这种对结构和类的支持, 因为为了遵循封装的原则, 我很少输出数据结构。

XS 方案允许你修改 XS 文件并提供不同程度的粘连代码 (用 C 编写)。它与 C 或 Pascal 编译器允许你在程序中插入本地汇编代码类似。如果你知道自己是在做些什么的话, 它就能够提供许多强大的功能, 但是也要求你要熟悉内部的 Perl API 及协议。

通过修改 XS 文件, 你可以创建接受不定个数输入参数的 `write` 包裹函数, 改动一些输入参数 (象 `read` 所做的那样) 并返回包含结果值的数组。加上编写定制类型映射和修改 Perl 模块 (由 `h2xs` 产生) 的能力, 于是你就有了多种创建扩展的方式。

让我们来简单的浏览一下XS的语法。我们前面例子中的*Fractal.xs*的最基本样式就是下面这个样子：

```
#include <mandel.h>

MODULE = Fractal    PACKAGE = Fractal

int
draw_mandel (filename,width,height,origin_real,origin_imag,range,depth)
    char*  filename
    int    width
    int    height
    double origin_real
    double origin_imag
    double range
    double depth
```

所有位于 MODULE 语句前面的文本都被当作原始 C 代码，并会不经任何转换，发送到粘连代码 *Fractal.c* 中（这类似于 SWIG 中的 `%{ ... }` 块）。一个 XS 模块中可以包含不止一个包，但是由于这种情况并不常见，因此关键词 MODULE 和 PACKAGE 拥有相同的值。所有可输出的函数都以特殊的方式列出。首先是自己单独一行的返回值（在缺少返回值的情况下，你必须指定 void），接着是包含一组参数名的函数名，最后是每个参数一行的排列。将“\*”与类型而不是名字写在一起非常重要——你必须这么写 `char* filename`，而不是 `char *filename`。下一个函数声明简单的在一个空白行后开始。

## xsubpp 都完成了些什么工作呢？

能够对由 *xsubpp* 所产生的粘连代码有所了解是值得的。在提交给 *xsubpp* 上面所描述的 XS 代码后，它会创建一个与 XS 声明的原型相同的名为 *Fractal\_xs\_draw\_mandel*（在 *Fractal.c* 中）的函数。该函数将 Perl 空间中提供的参数转换成 C 函数的参数，调用真正的 *draw\_mandel* 函数，最后将返回值打包到一个 Perl 的值中。

XS 提供了几个关键词，用来在所生成函数的合适的位置插入你自己的代码，或者将生成的粘连代码整个的替换成你自己的。例如，你可以编写类型映射函数，处

理如何将 Perl 的参数转换成 C 的参数; 你可以使用关键词 CODE (后面将会讲到) 来指定你要提供自己的代码。

脑子里带着这个简单的概述, 让我们来看一下 XS 语言的几个重要方面。

## 默认及可选参数

同在 C++ 中一样, 参数可以有默认值, 但是只能应用在最右端的参数上。下面是你如何在接口文件指定函数的原型的例子:

```
draw_mandel (file,width,height,orig_real,orig_imag,range,depth=30);
```

它允许你在从 Perl 中调用时可以选择跳过最后一个参数。

## 参数的修改

XS 允许你在提交给真正的 draw\_mandel 函数之前进行对参数的修改:

```
int
draw_mandel (filename,width,height,origin_real,origin_imag,range,depth)
    char*  filename
    int    width
    int    height
    double origin_real
    double origin_imag
    double range
    double depth
INIT:
if (width > 400) {
    fprintf (stderr, "Width cannot exceed 400. Truncating.\n");
    width = 400;
}
```

关键词 INIT: 告诉 XS 将跟在它后面的代码插入到参数转换 (从 Perl 到 C) 与调用真正的函数之间。

在 SWIG 中, 你可以使用一个有名类型映射来达到同样的效果。然而 XS 方式还允许你根据一个以上的参数来做决定。例如, 如果你需要保持某种特定的方面比

例，你就必须查看宽度与高度并对其中的一个进行修改。类型映射则无法提供这种灵活性，因为它是分立的来看待每个参数的。

附带说一下，关键词 `PREINIT:` 可以被用来插入变量声明；`xsubpp` 将会把这些声明放在任何产生代码的前面。当然这个关键词对于使用 C++ 编译器来编译粘连代码的情况并不重要，因为它允许在代码的任何地方声明变量。

## 特殊代码

如果需要的话，你可以自己来编写粘连代码。考虑一下数学库中的 `sin()` 函数，它要求你提供以弧度的形式来提供角度。你可以使用关键词 `CODE` 在 Perl 中创建一个新的函数，用来接受度形式的角度，代码如下（缩进模式是任意的）：

```
double
d_sin(angle)
    double angle
    CODE:
        RETVAL = sin(angle * PI / 180);
    OUTPUT:
        RETVAL
```

当 `xsubpp` 看到 `CODE` 关键词后，它就会将参数从 Perl 的数据类型中映射到 C 类型，然后让你来提供剩余部分的代码，这就意味着你必须自己调用底层的外部子例程。`CODE` 指令本质上并不改变 C 调用的结构；你可以修改输入参数还可以至多返回一个结果值。

`OUTPUT:` 指令告诉 `xsubpp` 提供一些将返回结果打包，并加载回 Perl 空间的代码。`RETVAL` 由 `xsubpp` 自动加以声明用来匹配函数的返回值。在前面的例子中，`sin()` 的返回值是唯一的输出参数，它在 `OUTPUT` 下列出。

如果你需要一种不定个数的输入参数或返回结果，那么 `CODE` 指令就帮不了什么忙了。在这种情况下，你可以使用 `PPCODE` 指令并显式的管理整个参数栈。我们将在第二十章更详细的讨论这个问题。

请查阅 XS 文档来了解有关其他关键词的细节及例子。

## C++ 代码

XS 支持两种特殊的过程来创建和删除 C++ 对象。考虑下面模块 Car 的 XS 代码:

```
Car*  
Car::new()  
void  
Car::DESTROY()  
void  
Car::turn_left()
```

当你在 Perl 中写到 `new Car` 时, `Car::new` 相应的包裹代码将会调用 C++ 代码, `new Car()`。而后当你在 Perl 空间中写到 `$car->turn_left` 时, 相应的 C++ 函数将会被自动调用。如果你想为 C++ 接口提供 CODE 或 PPCODE 指令, 你可以使用 `THIS` 来指代这个对象, 以 `CLASS` 来指代该类。

这个例子有一个毛病。它并不了解数据类型 `Car` 中的内容。与 SWIG 不同, `xsubpp` 希望提供类型映射形式的帮助, 而前者则不考虑那么多, 只将 `Car*` 当作 `void*` 来看待。由于我们需要了解内部 Perl API 才能创建一个类型映射, 因此我们将这个问题留在第二十章来解决。

## 自由度

在我们结束本章实质部分的讨论之前, 让我们来快速回顾一下为了帮助 XS 和 SWIG 产生平滑的接口 (对脚本程序员来说), 你可以在哪些地方插入代码:

### Perl 模块

到目前为止, 由这些工具产生的 Perl 模块只被用来引导 C 代码, 但是没有理由不应该拥有定制的子例程。我们前面“参数的修改”一节中讲述的 XS 例子可以轻而易举的在 Perl 空间中实现。

### 类型映射

支持用于在 Perl 与 C 数据类型之间进行转换的代码段。

### 接口文件中的包裹代码

CODE与PPCODE指令允许你插入各种完成定制转换任务代码。在SWIG中你可以像下面这样嵌入定制C代码:

```
%module FunMath
%inline %{
    int factorial(int n){return (n == 1) ? 1 : n *(n -1)};
%}
```

## 分形介绍

如果没有一小节用以简单的讲述一下有关Mandelbrot集和draw\_mandel实现的内容的话,那么这一章就会是不完善和干瘪的。

对于初学者来说,我强烈向你推荐Ivars Peterson的书《The Mathematical Tourist》,它的内容引人入胜,并令人吃惊的描述了范围广泛的数学主题。我们一开始就假定你已经了解了有关复数的知识。

我们知道一个复数 $a+bi$ 是由实部 $a$ 和虚部 $b$ 两部分组成的,它们结合起来就反映为图上的一个点。现在,考虑表达式 $z = z^2 - 1$ ,这里 $z$ 是一个复数。我们首先从一个复数( $z^0$ )开始将它描绘出来,然后将它替换到上面的表达式中又产生一个新的复数,把这个数也画出来。这种操作重复执行,比方说执行了20或30次。我们会发现不同的初始值 $z^0$ 要么生成一系列无限发散的点集,要么就是受限于某一边界值。所有导致产生受限序列的 $z^0$ 都属于Julia集,它是以数学家Gaston Julia的名字命名的。换言之,如果我们将所有导致产生受限序列 $z^0$ 绘制出来,就会看到一个漂亮的分形图(当然不是我们前面所见到的那种)。

现在让我们使这个等式变得更一般化一些:  $z \leftarrow z^2 + c$ , 这里 $c$ 是一个复数(上面的讨论是就 $c = -1 + 0i$ 来说的)。现在如果我们对不同的 $c$ 值绘制出Julia集的话,就会发现一些绘制显示出美丽的连接起来的图形,而另一些则发散成一群云雾般的离散的点。显然我们只对前者感兴趣;所有导致产生如此美观Julia集的值 $c$ 都被称为属于Mandelbrot集,取名于Benoit Mandelbrot。



计算 Mandelbrot 集显然是一件痛苦的活儿，因为对每个  $c$ （一个无穷集），你必须绘制出 Julia 集来看一看它是否是发散的。这里要提到数学家 John Hubbard 和 Adrien Douady。他们证明了对于一个给定的  $c$  值，足以检测一个初始点  $z^0=0$ （也就是  $0 + 0i$ ）是否会产生受限序列。如果是的话，那么值  $c$  就会产生一个连接起来（非发散的）的 Julia 集。而且已经证明了所有属于 Mandelbrot 集的  $c$  都被保留在一小块“从一边看看有小丘疹的雪人”（Ivars Peterson 这么来形容）形的区域内。这就是图 18-3 内白色的中央区域，在  $x$  轴从  $-2$  扩展到  $+0.5$ ，在  $y$  轴上从  $-1.0$  扩展到  $+1.0$ 。因此，只要序列超出了 2，你就能知道它不是受限的，结果就表明  $c$  不会是 Mandelbrot 集的一部分。为了给图再增加一些视觉感染力，我们将试图为在观察窗口区域内的每个点赋值一种颜色，无论它是否属于 Mandelbrot 集。那些属于这个集合的点设为白色，而不属于其中的点就赋给一种灰色，这要看相应的序列要跳出边界有多远。

`draw_mandel`（它位于文件 *Fractal.c* 中，如例 18-3 所示）实现了前面描述的算法。下面解释一下其中的参数，那些产生图 18-3 的值在括号中显示：

*filename*

要产生的 GIF 文件名。

*width,height(400,400)*

以像素数为单位，GIF 图片的宽度和高度。

*origin\_real,origin\_imag(-1.4,1.0)*

最左上部的像素所对应的复数。

*range(2.0)*

复平面中所跨的宽度与高度。如果 *origin* 为  $-1.0 + 1.4i$ ，*range* 为 2 的话，图形的跨度就是从  $-1.0 + 1.4i$  到  $1.0 - 0.6i$ （ $y$  从上至下递减， $x$  从左到右递增）。如果你减小这个数值，画板就会针对复平面的更小区域。这样，*range* 就像一个放缩因子，图片会随该值增减以相反的程度发生改变。

*max\_iterations(20)*

在放弃检查之前应当针对  $z \leftarrow z^2 + c$  进行迭代的次数，以决定一个序列是否受限。

例18-3: mandel.c

```

#include <math.h>
#include <stdio.h>
#include <gd.h>
typedef struct {
    double r, i;
} complex;

int draw_mandel (char *filename,
                 int width, int height,
                 double origin_real,
                 double origin_imag,
                 double range,
                 double max_iterations)
{
    complex    origin;
    int        colors[51], color, white, x, y, i;
    FILE       *out;
    gdImagePtr im_out;

    origin.r = origin_real; /* 从左上部开始衡量 */
    origin.i = origin_imag;
    if (!(out = fopen(filename, "wb"))) {
        fprintf(stderr, "File %s could not be opened\n");
        return 1;
    }

    im_out = gdImageCreate(width, height); /* Create a canvas */
    /* 分配一些灰色系的颜色。从黑色开始，逐步统一递增 r,g,b
       的值。这有保持色调但变换亮度的效果。
       (黑色 = 0,0,0 而白色 = 255, 255,255) */
    for (i = 0; i < 50; i++) {
        color = i * 4;
        colors[i] = gdImageColorAllocate(im_out, color,color,color);
    }
    white = gdImageColorAllocate(im_out, 255,255,255);
    /* 针对画板上的每一个像素 ... */
    for (y = 0; y < height; y++) {
        for (x = 0; x < width; x++) {
            complex z, c ;
            int iter;
            /* 给定初始值与范围，将像素转换成一个等价的复数 c,
               范围值有着逆向缩放因子的作用 */

```

```
c.r = origin.r + (double) x / (double) width * range;
c.i = origin.i - (double) y / (double) height * range;

/* 检查上面计算出的每一个点, 看看重复代换
   到诸如  $z(\text{next}) = z^2 + c$  的等式中时, 是否仍然保持在一
   个有限的边界内。
   如果经过 <max_iterations> 次迭代它仍然没有超出
   白色的区域的话, 它就属于 Mandelbrot 集。
   但是如果超出了, 我们就根据序列要超出限定的
   远近来给它赋以相应的颜色 */
color = white;
z.r = z.i = 0.0; /* 起始点 */
for (iter = 0; iter < max_iterations; iter++) {
    double dist, new_real, new_imag;
    /* 计算  $z = z^2 + c$  */
    /* 大家知道  $z^2$  为  $a^2 - b^2 + 2abi$ , 如果  $z = a + bi$ , */
    new_real = z.r * z.r - z.i * z.i + c.r;
    new_imag = 2 * z.r * z.i + c.i;
    z.r = new_real; z.i = new_imag;
    /* 从 0,0 的毕达格拉斯距离 */
    dist = new_real * new_real + new_imag * new_imag;
    if (dist >= 4) {
        /* 位于 Mandelbrot 集的所有点都不会超出初始值
           2 个单位。如果它超出了边界, 就根据序列跳出 give
           边界的远近赋值给 'c' 一个有趣的颜色 */
        color = colors[(int) dist % i];
        break;
    }
}
gdImageSetPixel(im_out, x, y, color);
}
}
gdImageGif(im_out, out);
fclose(out);
return 0;
}
```



## 相关资源

1. SWIG. David Beazley.

可以自由从地址 <http://www.cs.utah.edu/~beazley/SWIG/swig.html> 处下载。SWIG 中打包有一份约200页的精彩教程风格的文档，其中包含了许多有趣的例子。你在那里浏览时，请看一看 Dave 的有关将 SWIG 应用到大型项目中的文章。

2. *perlxs*ut, Jeff Okamoto 及 *perlxs*, Dean Roehrich.

这两份标准 Perl 文档分别为 XS 提供了一份指导教程和一份参考手册。你必须要么已经熟悉了第二十章的内容，或是 Perl 的内部工作文档 (*perl guts*)。(前者是一种稍微简单一些的介绍。)

3. Perl 标准扩展。

随 Perl 发行版一起发行的 Socket, POSIX, 以及 SDBM 模块提供了有关 XS 应用的优秀案例。

4. XS Cookbook. Dean Roehrich.

这些编程指导可以从 CPAN 获得 (请在 *authors/Dean\_Roehrich* 目录下查找), 它们对许多覆盖所有 XS 功能的范例问题提供了解决方案。我向你强烈推荐。你也会发现使用 SWIG 来解决这些问题会是一种很好的练习。

5. *The Mathematical Tourist*. Ivars Peterson. W.H. Freeman and Co., 1988.

6. 用于绘制 GIF 文件的 GD 库. Tom Boutell.

下载地址为 <http://www.boutell.com/>。



本章简介:

- 为什么要嵌入?
- 嵌入概述
- 例子
- 增加扩展
- 相关资源

# 第十九章

## Perl 的嵌入: 简单的方式

当一个人怀着在大教堂中的心境去注视一堆石头时，它们就不再是一堆石头了。

—— 圣-艾克苏佩里 (译注 1)

如同我们有许多理由要为 Perl 编写 C 扩展一样，我们也有许多理由要从 C/C++ 中调用 Perl 脚本；我们称之为 Perl 解释器的嵌入 (*embedding*)。嵌入并不表示我们希望掩盖解释器的存在；它只是表示应用程序拥有全面的控制能力，并在需要时可以调用 Perl 的内部 API。

本章将介绍一种简单的将 Perl 解释器嵌入到 C 应用程序中的 API。这些函数并不标准（也就是说它们是在本书中引入的），它们可以使你无须了解 Perl 的内部细节、引用记数、内存管理以及调用规范。尽管我们将在下一节讨论这些细节问题，然而你无须了解它们也能完成有用的工作。由 Jon Orwant 和 Doug MacEachern 编写的 *perlembed* 文档对该主题提供了一种很好的指导教程风格的讲解，但是要求你要熟悉内部工作细节。

## 为什么要嵌入？

一个 C 应用程序可以以各种方式利用脚本语言：

---

译注 1： 圣-艾克苏佩里 (1900-1944)，法国著名作家。

### 利用用脚本扩展的强大功能

Emacs, Microsoft Office 和 AutoCAD 这样的应用程序都提供了脚本语言前端。尽管它们自身就工作的非常好,然而它们的真正能力则来源于编写脚本扩展的庞大开发群体。用 Brian Kernighan 的话说就是,能以连它的开发者都从没想到的方式进行运用的工具才是好工具。例如 Emacs 中的包 *calc* 就可以进行符号数学运算。谁会想到把它放在一个文本编辑器中呢(注 1)?

### 作为粘连工具

Emacs 就是这样一个优秀的例子,为了获取速度和与操作系统的接口,它用 C 语言实现了基本功能,而用 LISP 实现了其他的一切(它包含有一个嵌入式的 LISP 解释器),LISP 为 C 代码提供了必要的粘连机制。少了一些关键的 LISP 代码,编辑器甚至不能启动。

### 利用脚本语言的强大功能

我曾经有一次工作在一个需要与主机进行通信的基于 Unix 的项目中。从主机下来的文件需要进行奇特的格式化处理,它们当然不符合规范的要求。由于要对文件格式进行复杂的处理时,使用 Perl 要比使用 C 容易的多,因此我使用 Perl 脚本和一个嵌入式 Perl 解释器来分析这些文件,这样我就能随意的改变分析策略。

我本来可以选择使用一种更为简单的方法,那就是使用 `system(3)` 或 `popen(3)` 产生一个子进程,来执行外部的 Perl 脚本程序,并从一个临时文件或管道中获取它的输出。这种方案对于大量的应用来说,都工作的挺好,CGI 的成功应用就是一个典型。对于将应用程序分成两个分立的单独调试程序这种问题还有许多值得探讨的东西。但是对于我的应用来说它还不够快。此外流过接口的数据并不那么简单,因此这样一来我就得在一端编写大量的代码来完成格式化工作,并在另一端进行分析。创建子进程来执行外部脚本程序还有一个问题就是,它不能为你提供一个持续的上下文环境。也就是说,你每次启动一个 Perl 脚本程序,它都无法记住从上一次启动以来所做的任何事情,而且还要重新打开套接字连接、数据库连接,重新开始事务处理等等。Apache Web 服务器就采用了嵌入解释器的方式。

---

注 1: 当然,使用 vi 的伙计们就会问为什么了?

### 更好的 C 代码

编写一个脚本前端将迫使你简化接口函数以易于同脚本语言的集成。更可喜的是它还更利于其他程序员来使用你的功能库。

### 作为测量器

脚本工具提供了通过编程来存取嵌入在代码中的测量探测部件(用来监视性能、内存使用情况、动态断言等)的机会。例如,你可以在用户数量超过 50 时,自动的设置对所有用户的人站连接进行审计追踪。

### 强大的配置文件

简单的配置文件或许并不能满足应用的要求(例如那些具有名字-值属性的配置文件,如 Microsoft 所提供的 Windows 注册表)。

## 解释器嵌入概述

看起来也许很奇怪,嵌入 Perl 解释器与扩展 Perl 不同,竟然没有任何工具来自动的完成这项任务。为什么会是这样呢?要知道扩展也必须要考虑 Perl 与 C 之间的数据转换(输入与输出参数)。原因就在于 Perl 驱动 C 代码时,它能够准确的指定如何以及什么时候要加载一个 C 扩展。作为一名扩展编写人员,你的工作就是只是编写回调方式的 XSUB,并提供一些初始化操作;当脚本恰当的调用相应的函数时 XSUB 就会被调用。与之相比,由于不存在标准的编写 C 应用程序的方法,你必须决定何时初始化嵌入式 Perl 解释器,以及如何将控制传递给 Perl 脚本程序。

为了简化解释器的嵌入,本章将向你提供一种建立在 Perl 内部 API 之上的简单易用的包裹层。这些子例程专为本书编写,使你免除了要消化吸收超过 50 页内部工作文档的烦恼。但是如果你是那种刨根问底的人,那么第二十章“Perl 的内部工作”就为你提供了所需的资料。它还解释了这些“便利”子例程的代码。

巧的是 Perl 可执行程序也是由两部分组成的:一个就是包含核心 Perl 子例程的功能库(注 2)(在 Unix 上为 *libperl.a*,在 Microsoft Windows 系统中为 *perl.lib*,或者为二者相应的动态连接库形式),另一个就是一个简单的驱动程序文件,

---

注 2: 不要与 Perl 发行版中的 *lib* 目录搞混了。

*perlmain.c*, 其中包含了 `main()` 函数, 这个文件在去掉所有与可移植相关的部分后的内容如下所示:

```
#include <EXTERN.h>
#include <perl.h>
static PerlInterpreter *my_perl;
int main(int argc, char **argv, char **env)
{
    my_perl = perl_alloc();
    perl_construct(my_perl);

    perl_parse(my_perl, xs_init, argc, argv, env);
    perl_run(my_perl);

    perl_destruct(my_perl);
    perl_free(my_perl);
}
```

`perl_alloc` 与 `perl_construct` 创建了一个解释器对象。`perl_parse` 完成进一步的初始化工作, 分析通过 `argc` 与 `argv` 提供给它的命令行参数, 调用初始化例程 `xs_init` 来加载其他的扩展模块 (或者至少是初始化动态加载器), 最后是分析作为命令行一部分提供的脚本程序。`perl_run` 执行这个脚本程序。最终, `perl_destruct` 和 `perl_free` 关闭并释放解释器。

为了利用 Perl 的强大功能, 你所要做的一切就是将 Perl 功能库连接到你的应用中, 并克隆 *perlmain.c* 中关键部分的代码。我们将在本章后面“增加扩展”一节再来讨论 `xs_init` 的问题。直到那一节以前, 我们都假定不需要任何扩展, 并且将 `NULL` 而不是 `xs_init` 传递给 `perl_parse`。一旦 `perl_parse` 工作完成以后, 解释器就全部初始化完毕, 随后你就可以调用所有由 Perl 功能库输出的函数。但是这一章我们将只限于讨论表 19-1 中所列出的几个高级调用。

表 19-1 简化解释器嵌入的 Perl API 调用

函数名	描述
<code>perl_call_argv(</code> <code>char *sub,</code> <code>I32 flags,</code> <code>char **argv);</code>	该调用存在于标准 Perl 发行版中。它用一个以 <code>NULL</code> 结尾的字符串数组为参数调用一个子例程。不幸的是, 它返回结果的方式并不方便。因此我们将在本章中使用的唯一一个标志就是 <code>G_DISCARD</code> , 它用以告知 Perl 默默的丢弃所有返回结果。



表 19-1 简化解释器嵌入的 Perl API 调用 (续)

函数名	描述
<pre>perl_call_va (     char *sub,     [char *type, arg,]*     ["OUT",]     [char *type, arg,]*     NULL );</pre>	<p>这个调用提供了一种方便的接口，你需要将以空字符结尾的包含类型参数的列表传递给 Perl 子例程，然后将返回结果收集到一个参数列表中（与 printf 和 scanf 类似）。参数 type 可以是 i, s, 或 d（整数，字符串或双精度浮点数）。字符串 OUT 表示一组返回参数的开始，它们是指定符与相应的类型的指针对。字符串输出参数被拷贝到所提供的缓冲区中，所以它一定要有足够的容纳返回字符串的空间。</p> <p>参数列表必须总是以 NULL 结尾。函数在失败时返回 -1，成功时则为过程所返回的参数个数。</p>
<pre>int perl_eval_va(     char *str,     [char *type,       *arg],     NULL);</pre>	<p>计算一个任意的字符串而不仅仅是子例程。字符串后面跟有如我们上面所讨论格式的任何数量的输出参数。它不需要输入参数，因为它们已经编码在字符串中了。perl_eval_var 在失败时返回 -1，或者为由计算所返回的结果参数的个数。</p>
<pre>set_int(char *var,         int value); int get_int(     char *var,     int *pvalue);</pre>	<p>获取或者设置一个可全局存取的、整数值的标量变量。var 中可以包含普通的标量变量名或数组和散列表索引如：foo, foo[10] 或 foo{hello}。</p> <p>get_int 获取了一个指向整数值的指针，如果成功返回 1，失败返回 0。</p> <p>set_int 创建一个变量，如果它原本不存在。</p>
<pre>set_double(char *var,             double             value); int get_double(     char *var,     double     *pvalue);</pre>	<p>与上面的类似。</p>
<pre>set_str(char *var,         char *value); int get_str(char *var,             char **value);</pre>	<p>Get_str 返回字符串的地址。你需要将它拷贝到自己的缓冲区中。</p>

Get\_\*和set\_\*函数只能被用来每次操纵一个标量变量。我之所以允许这种限制的原因就是，Perl中已经提供了许多很好的用来完成对数组及散列表的分段、切块和迭代操作的函数。我们将在第二十章再来详细的进行了解。这些函数尽管速度更快也更加精细，但仍然与内部工作细节（内存管理、临时变量等等）联系紧密，因此对它们的任何讨论都不可避免的要涉及其他方面。函数get\_\*和set\_\*则要简单的多。

## 例子

现在让我们编写一些代码来看看这些API是如何应用的。假定你有一个名为search.pl的Perl脚本程序，其中包含了子例程search\_files，定义如下：

例 19-1: search.pl

```
# search_files - 一个简单的grep。调用方法为……
#   search_files ("struct", "*.h")
sub search_files {
    my ($pattern, $filepattern) = @_;
    local (@ARGV) = glob($filepattern);
    return unless (@ARGV);
    while (<>) {          # 由于@ARGV已经初始化过了，所以可以这么做
        if (/ $pattern/o) {
            print "$ARGV\[$.\\]: $_"; # 文件、行号、匹配行
        }
    }
}
```

search\_files接受两个字符串参数而且没有返回值。我们有几种方式从C中调用这个子例程。让我们先从perl\_call\_argv()开始，因为它接受的是字符串参数。例19-2中的代码搜索所有C头文件中的单词“struct”。

例 19-2: ex.c: 嵌入Perl解释器

```
#include <EXTERN.h>
#include <perl.h>
static PerlInterpreter *my_perl;
main(int argc, char **argv, char **env) {
    char *my_argv[] = {"struct", "*.h", NULL};
    my_perl = perl_alloc();
    perl_construct(my_perl);
    perl_parse(my_perl, NULL, argc, argv, env);
```

```
perl_call_argv("search_files", G_DISCARD, my_argv);

perl_destruct(my_perl);
perl_free(my_perl);
}
```

我们传递给 `perl_parse` 的是 `NULL` 而不是 `xs_init`，表示我们不想加载任何扩展模块。此外，我们没有调用 `perl_run`，而是使用 `perl_call_argv` 来调用 `search_files`（我们使用标志 `G_DISCARD` 来表示丢弃所有返回值）。下面是我如何在一台 Linux 机器上编译并连接这段代码的（注 3）：

```
% gcc -o ex -I/usr/local/lib/perl5/i586-linux/5.004/CORE \
      -L/usr/local/lib/perl5/i586-linux/5.004/CORE \
      -Dbool=char -DHAS_BOOL \
      ex.c -lperl -lm
```

我们已经创建了第一个定制的 Perl 解释器。由于我们将所有的命令行参数都传递给了 `perl_parse`，因此 `ex` 可以像 Perl 一样进行调用，如下所示：

```
% ex search.pl
```

它将会输出下列信息（当在 Perl 源文件目录中进行调用时）：

```
av.h[10]: struct xpvav {
cop.h[58]: struct cop {
cop.h[60]:      char *   cop_label;      /* 此结构的标签 */
cop.h[75]: struct block_sub {
cop.h[98]:      { struct block_sub cxsub;
...

```

我们需要将脚本名作为参数来传递，因为 `perl_parse` 对于给定的参数不进行任何转换。

我们还可以使用另两种调用来代替 `perl_call_argv`，如下所示：

---

注 3： 你不必记住或查找包含文件和函数库目录的路径。本章最后一节讨论了一个称为 `ExtUtil::Embed` 的模块，它使得创建嵌入式解释器轻松迅捷。

```
perl_eval_va("search_files (qw(struct *.h))",
             NULL);           // 没有返回参数
```

或者:

```
perl_call_va ("search_files",
              "s" , "struct",    // 字符串类型的第一个参数
              "s", "*.h",       // 字符串类型的第二个参数
              NULL);
```

显然perl\_eval\_va方法对于这个例子来说是最容易的。附带说一下,你注意到了我们如何使用qw操作符来避免嵌入引号的吗?

我们现在再来看另外一个需要混杂传递多种参数类型的小例子。这一次我们将调用一个Perl子例程nice\_number,它在大数字间插入逗号(1000000被格式化为“1,000,000”)。下面的子例程每看到一组四个连续的数字时就插入一个逗号,并不停的进行这种操作直到没有匹配的这种模式为止。为了测试这个子例程,我们使用了一个附加的名为test\_nice的子例程,它会在提供给一个数字 $n$ 时,产生一个由1组成的 $n$ 位数字,然后将该数字提供给nice\_number使用:

```
sub nice_number {
    my $num = shift;
    1 while ($num =~ s/(.*\d)(\d\d\d\d)/$1,$2/g);
    $num;
}

sub test_nice {                      # test_nice(4)会产生1,111
    my $len = shift;
    nice_number(1 x $len);
}
```

我们没有将这段代码放到一个文件中并使用perl\_parse来进行分析(就像我们前面所做的那样),而是使用perl\_eval\_va来分析和加载这个子例程。但是perl\_parse要做一些关键性的初始化工作,因此我们还必须要调用它(注4)。如果我们提供给它一个空的argc/argv数组,那么很不幸,它会在标准输入上等

注4: 实际上,至多只能调用一次perl\_parse,因为它不检查是否已经做过而重新对解释器进行初始化。

候，正像你通常情况下期望 Perl 所做的那样。因此，我们提供给它一个尽可能短小的脚本，使它不存在任何编译问题，而且不费什么时间，如下面的命令行所示：

```
perl -e 0
```

现在唯一可能将这个脚本程序缩短的方法，就是减小其字体了！请注意例 19-3 中对 `perl_parse` 的调用：

例 19-3: ex2.c: 嵌入 Perl 解释器

```
#include <EXTERN.h>
#include <perl.h>
static PerlInterpreter *my_perl;
int main(int argc, char **argv, char **env) {
    static char *dummy_argv[] = {"", "-e", "0"}; int num;
    my_perl = perl_alloc();
    perl_construct(my_perl);

    perl_parse(my_perl, NULL, 3, dummy_argv, env);

    if (perl_eval_va (                                # 定义内联代码
        "sub main::nice_number {"
            "my $num = shift;"
            "l while ($num =~ s/(.*\\d)(\\d\\d\\d\\d)/$1,$2/g);"
            "$num;"
        "}"
        "sub main::test_nice {"
            "my $num = shift;"
            "nice_number (1 x $num);"
        "}",
        NULL ) == -1) {
        fprintf (stderr, "Eval unsuccessful. Aborted\\n");
        exit(1);
    }
    # 子例程已经定义。现在调用 test_nice
    for (num = 1; num <= 7; num++) {
        char buf[20];    *buf = '\\0';
        perl_call_va ("test_nice",
            "i", num,                # 输入参数
            "OUT",
            "s", buf,                # 输出参数
            NULL);                  # 不要忘了这一点！
        printf ("%d: %s\\n", num, buf);
    }
}
```

```
    }  
    perl_destruct(my_perl);  
    perl_free(my_perl);  
    return 0;  
}
```

打印输出结果为:

```
1: 1  
2: 11  
3: 111  
4: 1,111  
5: 11,111  
6: 111,111  
7: 1,111,111
```

## 增加扩展

在前面的章节中,我们创建了C应用程序,让它们调用Perl功能库、分析脚本并在Perl与C空间之间传递数据。在此期间,我们特意避免涉及扩展的问题。你也许还记得,我们传递给perl\_parse的是NULL而不是一个初始化子例程的地址。这就意味着我们在脚本中无法利用任何基于C的扩展,即便是那些常用模块,如Socket和SDBM——这对于实际应用来说显然无法接受。

在这一节,我们将学习一种使嵌入式Perl解释器能够存取标准或定制扩展的简单方法。

初始化子例程,我们称之为xs\_init,将负责为所有静态连接的扩展调用初始化例程。如果你倾向于使用动态加载的话,xs\_init就只需简单的初始化内建的动态加载器。

我们不是手工编制xs\_init代码,而是使用一个非常方便的名为ExtUtil::Embed的模块为我们产生一个。该模块与Perl发行版打包在一起,其使用如下:

```
perl -MExtUtils::Embed -e xsinit -- -o xsinit.c -std IO::Socket DBI
```

`-M` 选项等价于使用 “`use ExtUtils::Embed;`”。这个调用将会产生一个名为 `xsinit.c` 的文件，其中包含了一个公共可用的函数 `xs_init`，而它中间又包含了用于初始化所有标准模块（多亏了 `-std` 参数）及两个定制模块 `IO::Socket` 和 `DBI` 的代码。

这个模块怎么会知道什么才是标准的，怎么会知道我们想要将这些包以静态还是动态进行连接呢？其实，当 Perl 被编译并安装时，它会保存所有静态连接扩展（如果有的话）以及提供给 `configure` 脚本的参数，如编译连接选项，Perl 的安装地点等等的清单。这个清单保存在一个名为 `Config.pm` 的模块中。Embed 模块将抽取这些信息来产生合适的初始化工作集。此外，Embed 还可以被要求打印出编译和连接选项，我们可以像下面这样在命令行上加以利用：

```
% cc -c xsinit.c      `perl -MExtUtils::Embed -e ccopts`
% cc -c ex.c          `perl -MExtUtils::Embed -e ccopts`
% cc -o ex ex.o xsinit.o `perl -MExtUtils::Embed -e ldopts`
```

除了省去了我们手工编制初始化代码，以及填充恰当的编译连接命令行选项的麻烦之外，该模块还简化了将来增加其他的扩展模块的工作。当然，如果嵌入式解释器被设置为动态加载方式，那么就没有必要重新创建 `xsinit.c`，因为它里面只包含了一个初始化动态加载器的调用。

## 相关资源

1. `perlembed`（标准 Perl 文档）。

Doug MacEachern 和 Jon Orwant 撰写。

2. Apache 和 `mod_perl`。地址为 <http://www.apache.org/>。

Apache 是一种可以自由获得的 Web 服务器，它可以嵌入由 Doug MacEachern 编写的 Perl 模块 `mod_perl`，该模块提供了 Apache C API 与 Perl 之间的连接，可以允许你编写脚本操纵 Apache API，从而代替 CGI 脚本程序。据称与 CGI 方式相比，可在速度上有 400%~2000% 的提升。

## 第二十章

# Perl 的内部工作

### 本章简介:

- 阅读源代码
- 体系结构
- Perl 的值类型
- 堆栈与消息协议
- 内涵丰富的扩展
- 简单的嵌入式 API
- 未来展望
- 相关资源

它不能看见，不能触摸，  
不能听见，不能闻到。  
它呆在星星后面，山坡的下面。  
栖身在空空的洞穴中。(注1)

——J.R.R. Tolkien (译注1)

*《The Hobbit》*

本章我们要适度的涉及与Perl解释器有关的大多数关键数据结构和函数。对这些细节(的确很枯燥)的掌握，将会使你有信心编写功能强大的扩展模块，并使你有能力判断在具体的应用中应该怎样(以及在多大程度上)来使用Perl。一位好程序员的标志就是，能够回答那些并非出现在常见问题列表(FAQ)中的问题，比如下面的这些问题：

- 为什么对象要比闭包更为可取？
- `my` 为什么要比 `local` 速度快？
- 上一章所讲述的方便嵌入式API并不怎么方便。应该如何来编写自己的这种API呢？
- `xsubpp` 与 SWIG 究竟都产生了些什么东西？
- 为什么不能让Perl解释器输出Java字节码，也加入Java革命的潮流呢？

注1： 答案：黑暗。

译注1： J.R.R.Tolkien (1892 - 1973)，英国著名学者、作家。他对中古北欧诸语种及中古北欧神话、历史造诣极深。所撰写的《The Hobbit》和《The Lord of the Rings》两书流传很广。



等等。你所需要的就是对C语言的熟练掌握，一个爱问问题的脑筋和一把舒适的椅子。

如果你想立刻获得满足感，并等不急要制作出一个很酷的扩展，你可以选择阅读本章中的精简部分，即按照下面的顺序进行阅读：“Perl的值类型”，“堆栈与消息协议”以及“丰富的扩展”。你完全可以跳过所有标题为“……内幕”的章节而不会破坏连贯性。

## 阅读源代码

故事是这样的，有一个程序员对一段代码百思不得其解。这段代码根本就没有任何注释，而且他无论如何也搞不清它到底是怎么完成工作的。几年中他一直咒骂写这段程序的作者，但是这段代码依然吸引和困扰着他。有一天，他突然灵机一动，豁然开朗。其实，这段代码的意思是如此的显而易见，他甚至理解了为什么会没有任何注释！

尽管Perl的源代码或许是解决所有问题最终的原料库，但是让人相当的难以接受。缺乏注释，大量的使用宏以及一些令人吃惊的优化措施，使得理解源代码是一件相当艰苦的工作，即便是对于那些死硬派也是如此。如果你是那种喜欢探索和弄懂所有各种精髓的人，本章可以向你提供足以开始工作的知识。此外，这里还有一些能够更好的理解系统的方法：

### -D 选项

你可以选择使用 `-DDEBUGGING` 选项来编译Perl，这样会激活 `-D` 命令行开关。该开关选项可以接受好几种标志，所有这些标志在文档 *perlrun* 中都有介绍。就像进行CAT扫描一样，这些标志为运行时一些重要的数据结构提供恰当的快照信息。例如，以 `perl -Dts` 方式调用Perl时，就表示要显示操作码的执行路径（`-t`），以及在每个操作码执行前要打印出其参数堆栈（`-s`）。

### Devel 工具

这些模块可以从CPAN的 *Devel* 一层下获取，它们提供了对一些重要数据结构脚本级的存取访问。这些模块包括 `Devel::Peek`（用于打印出与某个变量相

关的内部信息), `Devel::Symdump` (用于打印出符号表) 以及 `Devel::RegExp` (用于检查正则表达式)。我们在本章中会经常用到 `Devel::Peek` 模块。

*调试器 (gdb、dbx、Microsoft Developer Studio)*

在调试器下查看 Perl, 可以提供整个进程的第一手观察资料。在运行时, 进程要经历三个主要的阶段: 初始化, 语法分析, 与执行; 这些都可以进行独立的检查。我建议你要首先理解 Perl 的值类型和堆栈协议, 然后通过 `run.c:runops` (注2) 处设置断点来试图理解执行阶段, 接着就继续向下考察。这个工具中最为复杂的部分就是语法分析器和代码生成器; 我建议你只有在对系统的其余部分有相当的了解之后再来试图理解这些东西。附带说一下, 像 `cxref` 这类工具帮不了太大的忙, 因为大多数令人感兴趣的存取操作都被宏、类型转换和间接指针掩盖起来了, 因此通过源码级调试器进行单步执行常常是唯一的选择。

本章要频繁的提到源代码文件, 虽然手头上准备好这些东西对你来说会方便一些, 但这又不是绝对必需的。

## 体系结构

图20-1描述了一个正在执行的 Perl 系统的各种部件。带深色阴影的矩形表示数据结构, 其中的一些在程序中还可以有多个实例。Perl 的源代码也可以粗略的沿这些线划分成几部分。

## Perl 对象

图20-1中标有“Perl 对象 API”的方框表示用于操纵所有内部数据结构 (如变量、符号表、堆栈) 和资源 (如文件与套接字) 的 API。

### 变量

我们在第三章“Typeglob 与符号表”中已经知道了“变量”这个字眼指的是一个名字-值对。在这一章中, 我们将来看一些 API, 它们用于操纵不同类型的值并有选择的将这些值与名字进行绑定。这些值是下面的一种:

---

注2: 我以前确实是针对那些 (技术上) 特别较真的人提出的。

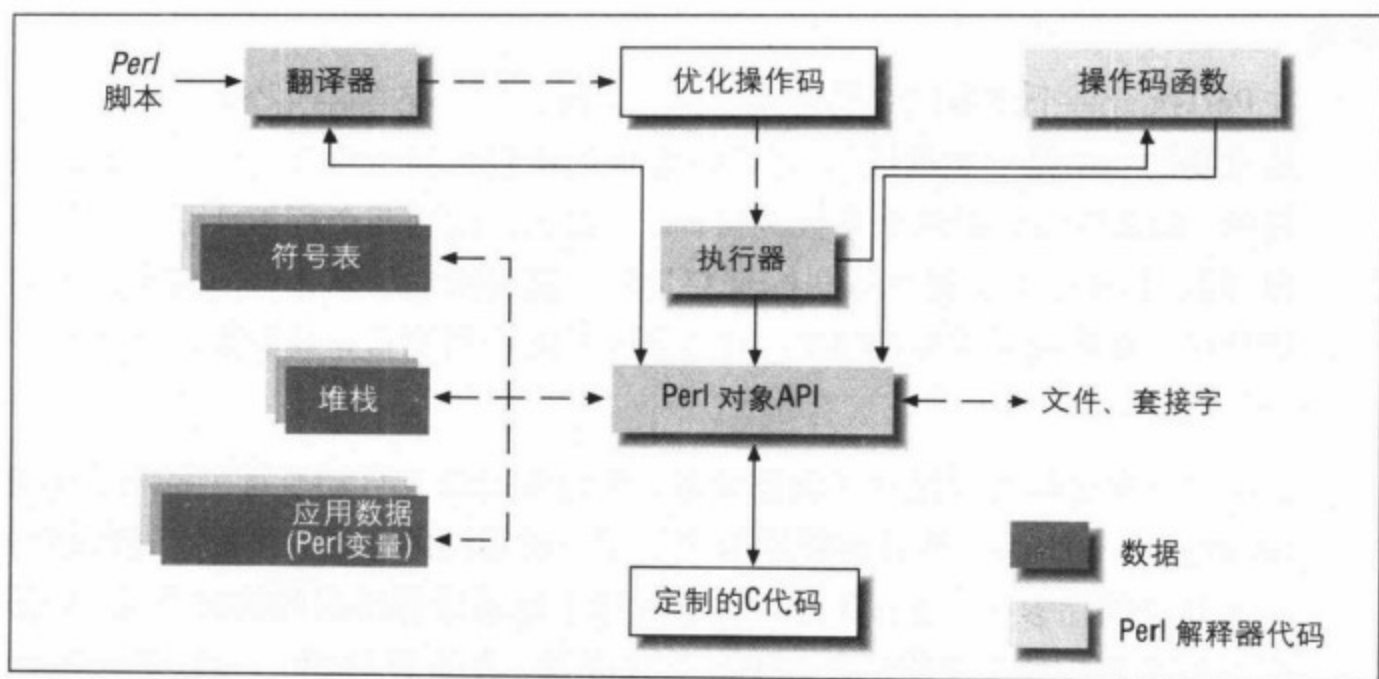


图 20-1 一个正在运行的系统的快照状态

SV: 标量变量值

AV: 数组值

HV: 散列表值

CV: 代码值

GV: Glob 值 (或 typeglob)

RV: 引用值

SV 可以进一步被分为 IV (整数值), PV (字符串值) 或 NV (双精度浮点值)。这些缩写是统一命名约定的一部分, 因此, 举例来说, 你可以很容易的猜出 `newSViv` 这个函数的用途来。

这些值类型提供了一种简单的 API, 能够自动的调整自己所占空间的大小并遵从简单的内存管理协议。因此, 大多数 Perl 的内部数据结构, 如堆栈和符号表等, 也都是依据这些值来实现的。

### 符号表

符号表就是普通而传统的 HV。它们的键值为标识符名 (字符串), 而它们的值为指向 GV 的指针。可原来不是说散列表的值应当为标量变量吗? 我们将在 “Glob 值与符号表” 一节中来回答这个问题。

### 堆栈

在Perl执行一个脚本程序的时候，它将运行时信息保存在几个堆栈中，其中最重要的一个是“参数栈”，在Perl源代码中也可简单的称之为栈。思想很简单，如果foo要以两个参数调用bar，它就会把这两个标量变量压送到栈顶再调用bar。bar提取这些标量变量后，就去做它的工作并将结果返回到堆栈中。堆栈就是简单的AV，对每个嵌套的调用而言，其参数都会占用一段栈空间。

C程序员将堆栈当作保存子例程参数、临时变量以及自动变量（局部于作用域的变量）的场所。Perl解释器则实现了一种不同的模式。上面描述的堆栈只保存子例程参数；还有其他一些堆栈用于记录计算得出的临时变量、局部变量和其他各种类型的信息如循环迭代变量，在遇到last、redo或return时要执行的下一个操作码等等。“栈与消息传递协议”一节中将有更详细的介绍。

### I/O 抽象层

Perl在内部使用一个称做PerlIO的对象来处理所有的I/O需求。这层抽象实质上是建立在两个功能库（即stdio和另一种速度要快的多的sfio）上的薄薄的可移植层。我们在这一章并不会讨论I/O抽象层的问题，因为它很简单而且没有什么高深的内涵。请阅读perlapi文档了解其细节。

### 多个解释器

上面所描述的数据结构通常保存在全局C变量中。如果在编译Perl时使用了-DMULTIPLICITY选项，那么就会将这些全局变量打包到一个称作PerlInterpreter的数据结构中。这样就允许你同时拥有多个解释器的实例，每一个都有它自己的“全局”空间。（回想一下第十九章中用于分配和构建PerlInterpreter类型对象的API。）如果没有这个编译时选项，那么PerlInterpreter对象就是一个哑结构，而内部数据结构将会具有真正的全局性，从而可以获得最高的执行性能。API在任何一种情况中都是一样的。

你可以使用多个解释器来强制得到完全分离的名字空间。每个解释器都有它自己的包“main”以及自己已加载的包的树形组织。我没有见过在Perl应用系统的制作中使用这项功能，但是Tcl提供了一种称为SafeTcl的框架应用于安全目的，它

使用的一种类似功能就具有多个解释器对象。Perl中与之对等的模块 Safe 尽管结果类似（也具有分离的名字空间），但却使用了一种不同的机制。我们将在下一节更多的谈到这个问题。

## 翻译器

翻译器（translator）将 Perl 脚本程序转换成操作代码树（下面再解释）。它包括一个手工编制的词法分析器（*token.c*），基于 yacc 的语法分析器（*perly.y*）以及代码生成器（*op.c*）。形成了一种独特子语言的正则表达式，可以在 *token.c* 中识别，并会由文件 *regcomp.c* 中的代码编译成一种内部格式。

操作码与机器码的概念类似；机器代码是由硬件执行的，而操作码（opcode，有时被称做字节码或 p-code）则由一个“虚拟机”执行。它们的共同点也就只有这么多了。由于性能的原因，现代的解释器决不会模仿硬件 CPU 的工作。相反，它们为执行创建复杂的数据结构，这样每个操作码运行时都直接包含指向下一个要执行的操作码以及指向所需工作数据的指针。换句话说，这些操作码不只是指令类型；它们实际上包含了程序中那一刻所需要执行的工作单元。

Java 与 Perl 均是这种解释器的实例。许多 Java 字节码模仿一种 RISC 机器指令集，而 Perl 的操作码则代表一种更高级的抽象。大量的这些操作码直接与脚本级别的可用工具相对应，如正则表达式，匹配与替换，*chop*、*push*、*index*、*rindex*、*grep*（注 3）等，这也解释了为什么在写这本书时会有 343 种操作码的原因！还解释了 Perl 的速度为什么会是这么快：它没有在解释器中花费时间，大多数的工作都是在地道的手工编写的 C 代码中完成的。你也能够明白为什么编写一个 Perl-Java 的字节码翻译器会如此困难的原因：在这两种操作集之间没有任何相符之处。

### 操作码内幕（注 4）

*op.h* 中定义了一个由所有操作码所共享的基本数据结构 *op*。其中在这一节中要讨论的重要字段有：

---

注 3： Perl 的 *grep* 运算符，而不是 Unix 工具。还没有到整个工具都由操作码来表示的程度！

注 4： 除非你有强烈的想了解内部工作情况的愿望，在读本章第一遍时你不必消化——甚至阅读这一节的内容。标题为“... 内幕”的章节是一些较为独立的片段。

```

OP*      op_next;
OP*      op_sibling;
OP*      (*op_ppaddr)();
OPCODE   op_type;

```

Op\_type字段包含了操作码的实际类型。opcode.h中提供了所有操作码类型的列表，它是由脚本opcode.pl在创建解释器时自动产生的。该脚本中包含了描述所有操作码的格式美观的列表，因此相对opcode.h来说，是更好的信息来源。

指针op\_ppaddr代表了操作码的实质性内容：它是一个指向内建函数的指针（称为操作码函数），由它实现操作码的功能。所有操作码函数均带有前缀pp（pp\_push、pp\_grep等等），并且以pp.c、pp\_ctl.c、pp\_sys.c以及pp\_hot.c的形式发行。最后一个文件包含了那些比较“热”，也就是执行频率高的操作码函数，因此它有可能被保留在大多数RISC系统的高速缓存中。Tom Christiansen曾经提到过，这项功能同样也适用于正则表达式匹配的代码，这也是为什么用Java编写的正则表达式匹配器在速度无法与Perl相提并论的原因。（我将会在Sun的Java处理器可以自由获取以后，再来重新考察这个论断。）正如你在后面将会看到的那样，操作码函数与xsubpp/SWIG的输出代码看起来如此的相似；这是因为它们使用参数堆栈来进行互操作，而且遵循相同的参数传递协议。

根据操作码类型的不同，它们还会有附加的结构成员。例如，add操作码是一个二元运算符，因此它会包含两个指向其子节点的指针，而且会在这些节点被计算以后才进行结果的相加。操作码print是一个列表操作符，因此它会包含一个指针，指向其子列表中的第一个操作码，而它又通过op\_sibling指针（由所有操作码所拥有）连接到它们的兄弟节点，等等。

这种复杂的相互连接的网络结构被称作语法树。图20-2就描述了这样一个将表达式print \$a + 2进行分析后产生的语法树。

语法树自顶向下的数据结构指示了表达式的优先级；子表达式\$a+2必须在打印开始前计算出\$a+2的值。这个记号还表示在获取\$a的值与常数2并推入堆栈以后，才能进行加法操作。这样操作码gvsv（它会获取\$a的值）以及操作码const就会是add操作码的子节点，而它们之间又是兄弟关系。操作码add依次又会是print操作码的子节点。正如你所看到的，这些子节点与兄弟节点指针所组成的网络就反映了程序的语法结构。

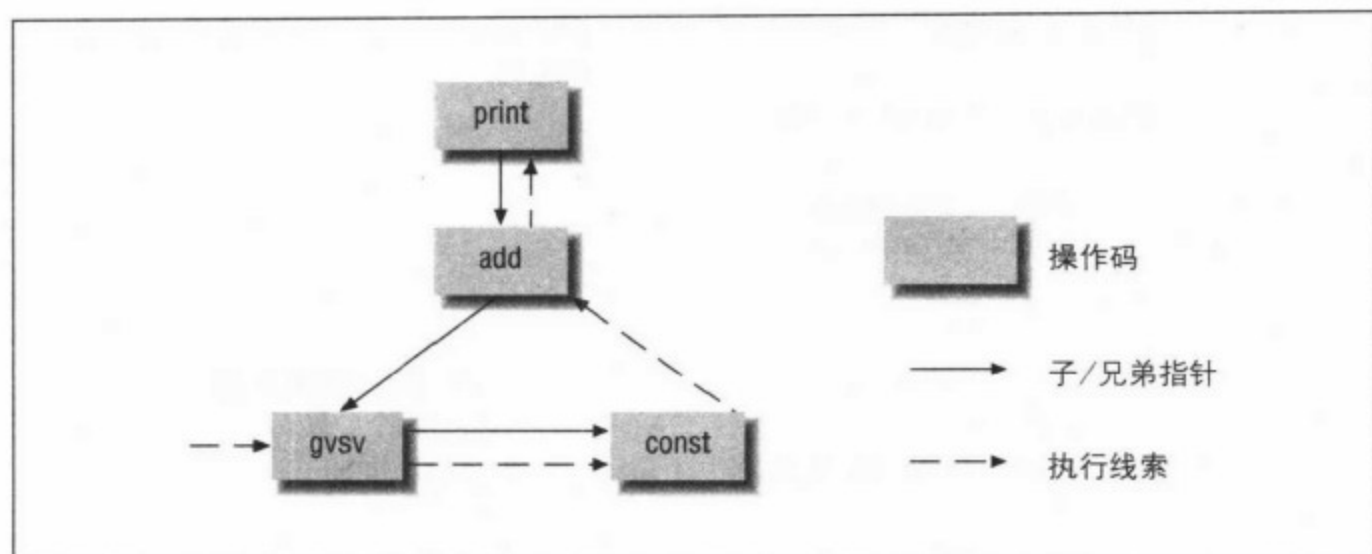


图 20-2 “print \$a + 2” 的语法树及执行线索

`op_next` 指针指向下一个要被执行的操作码也就代表了控制流向。这样执行代码就是简单的移动到下一个操作码并调用其操作码函数。图 20-2 的虚线表示了执行的线索。

如果你编译 Perl 时使用了 `-DDEBUGGING` 选项，你就可以使用 `-Dx` 命令行选项来告诉 Perl，在完成了对脚本的分析之后打印出它的语法树。例 20-1 描述了命令 `perl -Dx -e 'print $a + 2'` 的输出结果。嵌套层次反映了树形结构（见图 20-2 旁边的标注），记数模式则描述了执行的顺序。（例 20-1 中，我将注释添加在右边；其余的部分就是 Perl 的输出。）

例 20-1: “print \$a + 2” 的操作码执行顺序和树形层次结构；使用 `-Dx`

```
{
8  TYPE = leave ==> DONE          # 清除上一条指令
  FLAGS = (SCALAR, KIDS, PARENS)
  {
1    TYPE = enter ==> 2          # 从这里进入
  }
  {
2    TYPE = nextstate ==> 3      # nextstate 插入到每条语句之后
                                # 用以清除临时变量
    FLAGS = (SCALAR)
    LINE = 1
  }
}
```



```

7      TYPE = print  ==> 8      # 调用 print, 包含了首先要进行计算的
                                   # 表达式
      FLAGS = (SCALAR,KIDS)
      {
3          TYPE = pushmark ==> 4
          FLAGS = (SCALAR)
      }
      {
6          TYPE = add  ==> 7      # add 在参数堆栈顶部
          TARG = 1                # 需要两个参数
          FLAGS = (SCALAR,KIDS)   # (讨论如下)
          {
              TYPE = null ==> (5)
              (was rv2sv)
              FLAGS = (SCALAR,KIDS)
              {
4                  TYPE = gvsv ==> 5      # 获取与名字
                  FLAGS = (SCALAR)        # "main::a" 关联的简单变量值
          GV = main::a
              }
          }
          {
5              TYPE = const ==> 6      # 将常量 "2" 推入堆栈
              FLAGS = (SCALAR)
              SV = IV(2)
          }
      }
  }
}

```

每对花括号代表一个特定操作码的信息。第一个要执行的操作码为 `enter`。它然后将控制转交给 `nextstate`，而它又依次交给 `pushmark` 等等。操作码 `nextstate` 清除一个语句引入的所有临时信息，并为那个作用域中的下一条语句作好准备。当你在本章的后面学到参数传递协议时，`pushmark` 操作码的用途就会一目了然。

编译时，操作码 `gvsv`（用来获取全局或局部变量）将以其执行时要存取的值的地址，以及要推入堆栈的地址进行初始化。这意味着它在运行时根本无须再访问符号表——它已经掌握了所需的值。只有在你使用诸如符号引用、函数的动态联编以及 `eval` 等在编译时不能得到所有信息的功能时，才需要参照符号表。



## 编译及代码生成阶段

yacc以自底向上的方式工作,因此先产生处于语法树叶节点的操作码。随着分析的进行,处于语法树高层的操作码就会将它以下的节点连起来。每个一元和二元的操作码(如算术运算符)都会被检查是否可以立即执行;这被称做恒定折叠(*constant folding*)。如果可以执行,那个操作码和它的子节点将被删除,而且会在原位替换成一个const操作码。下一步就是验证对应于内建函数的操作码,是否拥有正确的参数个数和参数类型。

接着就是上下文关联(*context propagation*)。在创建时每个操作码都要为它自己及其子操作码指定一个上下文(void、Boolean、list、scalar或lvalue)。考虑表达式substr(foo(), 0, 1)。操作码代表一个对foo的调用,并首先创建常量0和1。接着在substr的操作码创建时,它会告诉代表对foo调用的操作码,它需要一个标量变量类型的结果。因此上下文关联是自顶向下工作的。

在分析完成以后,一个窥孔优化程序(*peephole optimizer*)开始执行(函数peep位于op.c中)。它沿指针op\_next跟踪所有的执行分支,就像在运行时执行的那样,搜索可以进行局部优化的地方。(也就是说它沿着执行路径试着执行了一遍。)这个过程通常检查紧接着的几个操作码(这里是至多两个),并检查是否可以减少成更简单的或更少的操作码;换句话说,它透过一个窥孔进行检查。让我们通过一个小例子来稍微详细地考察一下这个问题。

\$符后可以跟一个标识符名(\$a),一个数组元素(\$l[0]),或最一般的情况,一个结果为标量变量引用的表达式(\$\$ra或\$foo{()})。在第一趟扫描中,分析器假定一种最为通用的方式,因此即便是简单到\$a这样的东西,都会被简化成两个操作码:gv和rv2sv。第一个获取一个GV(一个typeglob,如果你想一下就会知道这是一个指向一个标量变量值的引用)并将它推入堆栈,而第二个操作码则将其转换成一个SV。接着就是窥孔优化程序的工作,它将这个序列替换成一个操作码gvsv,它一次就能完成相同的工作。问题就在于,删除这些没有用的操作码是一项费时且冗长的操作,因为这些操作码包含了对其他操作码的连接。因此无用的操作码只被简单的标志为空,而前面操作码的op\_next指针会简单的将它们(被置为空的操作码)跳过去。例20-1描述了将操作码置为空的例子,请查看TYPE=NULL的那一行(以前为rv2sv)。

## 安全功能

Perl 提供 `-T` 开关选项来激活污染检测模式 (*taint checking*)，它会将那些包含从外界程序导出的数据的变量，标志为受污染的。它实现了一种数据流机制，通过它，所有从这些变量导出的变量都被认为是受污染的。(将一个受污染的标量变量推送到一个数组中，就会将这些数组标志为可疑的。)这种方式本质上意味着，你托付给代码来正确完成工作，并让它来区分受污染的和未受污染的数据。但是如果代码本身是可疑的，你可以使用标准的 `Safe` 模块及其附属包 `Opcode` (注 5)。这些模块可以允许你创建一个安全隔间 (*safe compartment*)，并为其指定一个操作码屏蔽 (一个允许的操作码的列表)。你可以在这个隔间中 `eval` 一段不受信任的代码，而且如果该编译过程产生了一个不在操作码屏蔽中的操作码的话，它就会返回一个错误。在下面的几个版本中，预期 Perl 将会考虑到其他类型的恶意攻击，如不受限制的内存分配 (`@l=(1..1000000)`) 或 CPU 灾荒 (`1 while (1)`)。这些也被称做拒绝服务 (*denial of service*) 攻击。

## 执行器

执行器 (*executor*) (文件 `run.c` 中的函数 `runops`) 是一个简单的驱动程序，它将会遍历语法树的执行路径并顺序调用每个对应的操作码函数。但是由于 Perl 是一种动态语言，并不能够一开始就确定执行路径，因此每个操作码函数都需要返回下一个要执行的操作码。一般来说就是顺序中的下一个操作码 (在编译时设置的 `op_next` 指针)。但是诸如 `if` 这样的条件操作符，或 `$foo->func()` 那样的间接表达式，只有在运行时才能决定下一个操作码。

我们现在就结束对 Perl 体系结构的简单介绍。

## Perl 的值类型

在这一节，我们将学习用于操纵内部值类型的函数与宏。我们还将查看每个对象的内部组成，这些内容将在标题为“SV 内幕”，“AV 内幕”等章节中讲述。尽管

---

注 5： 这两个都是由 Malcolm Beattie 设计的 (请查看一下标准 Perl 库中子目录 `ext/Opcode` 下的内容)。

这些内容可以帮助你内存的开销及性能作出成熟的判断,但是如果这些细节使你不堪重负,那么你完全可以跳过去。

## 标量变量值

一个标量变量值 (SV) 包含有标量变量的值, 一个引用记数以及一个描述标量变量状态的位屏蔽。标量变量可以是一个整数值 (IV), 一个双精度浮点数 (NV), 一个字符串 (“PV” 表示指针值), 一个引用 (“RV”) 或是一种特殊目的的对象 (“魔术”)。我们将在另外的章节中讨论魔术 (magical) 变量。

表 20-1 中描述了用于创建、删除以及修改 SV 的函数和宏。它们是在文件 `sv.h` 中声明, 在 `sv.c` 中实现的。按照惯例, 宏的第一个字母为大写。这一章中的所有表格都使用了两种重要的 typedef: `I32` 和 `U32`, 分别表示有符号和无符号的整数值, 它们至少都有 32 位并且足以容纳一个指针 (在 64 位机器上它将会为 64 位)。

表 20-1 标量变量值的 API

函数 / 宏	描述
<pre>SV* newSViv(I32); SV* newSVnv(double); SV* newSVpv(char* str,              int len); SV* newSVsv(SV *);</pre>	<p>分别从整数、双精度浮点数以及字符串创建一个新的 SV。如果 <code>len</code> 为 0, 那么 <code>newSVpv</code> 将会计算出字符串的长度。</p> <p>创建现存 SV 的克隆。要创建一个空的 SV, 要使用全局标量变量 <code>sv_undef</code> 而不是 <code>NULL</code>, 例如:</p> <pre>newSVsv(&amp;sv_undef);</pre> <p>对于所有需要以 SV 为参数的函数均是如此。</p>
<pre>SV* newSVrv     (SV* rv,     char *pkgname);</pre>	<p>创建一个新的 SV 并使指针 <code>rv</code> 指向它。而且, 如果 <code>pkgname</code> 非空, 它会将 <code>rv</code> bless 到那个包中。</p>
<pre>SV *newRV (SV* other) SV* newRV_inc (SV* other) newRV_noinc(SV *)</pre>	<p>创建一个指向任何类型值而不仅仅是 SV 的引用。你可以将其他类型的值转换为一个 SV*, 在考察 SV*, AV, HV 以及 CV 时显然就会如此。</p> <p><code>newRV_inc</code> 将会增加所指向实体的引用记数 (而且是 <code>newRV</code> 的别名)。</p>

表 20-1 标量变量值的 API (续)

函数 / 宏	描述																				
<code>SvIOK(SV*), SvNOK(SV*), SvPOK(SV*), SvROK(SV*), SvOK (SV*), SvTRUE(SV*)</code>	这些宏将检查这些SV是否拥有相应的值类型, 如果是这样的话就会返回 1。它们并不进行任何转换。 SvOK 在值不是 undef 的情况下返回 1。SvTRUE 在标量变量为真的情况下返回 1。																				
<code>IV SvIV(SV*) double SvNV(SV*) char* SvPV(SV*,int len) SV* SvRV(SV*)</code>	这些宏将会获取 SV 中的值, 除 SvRV 以外, 在必要时会强行为隐式转换相应值。如果标量变量包含了一个非数字字符串, SvIV 就会返回 0。SvPV 返回一个指向字符串的指针并将len更新为它的长度。该标量变量拥有这个字符串, 因此不要释放它。在调用 SvRV 前, 要使用 SvROK 来确保它真的是一个引用。																				
<code>sv_setiv (SV*, int) sv_setnv (SV*, double)</code>	修改一个 SV 的值。SV 将会自动的清除掉它原先的值并转换成这个新的类型。																				
<code>sv_setsv (SV* dest, SV* src)</code>	在检查到这两个指针并不相同之后, sv_setsv将会把 src SV 拷贝到 destSV。																				
<code>sv_setpv (SV*, char *) sv_setpvn(SV*, char *, int len);</code>	这些是字符串函数, 它们在必要时会强制将标量变量转换为字符串。sv_setpv假定字符串的结尾为空值, 而 sv_setpvn则接受字符串的长度。这两个函数都将创建一个给定字符串的拷贝。																				
<code>sv_catpv (SV*, char*); sv_catpvn(SV*, char*, int); sv_catsv (SV*, SV*);</code>	cat 系列函数完成字符串的连接。																				
<code>SvTYPE(SV*)</code>	返回一个枚举值, 并与 ref 函数等价。下面是 sv.h 中列出的一些常用的值: <table><tr><td>SVt_IV</td><td>(Integer)</td><td>SVt_NV</td><td>(Double)</td></tr><tr><td>SVt_PV</td><td>(String)</td><td>SVt_RV</td><td>(Reference)</td></tr><tr><td>SVt_PVAV</td><td>(Array)</td><td>SVt_PVHV</td><td>(Hash)</td></tr><tr><td>SVt_PVCV</td><td>(Code)</td><td>SVt_PVGV</td><td>(Glob)</td></tr><tr><td colspan="4">SVt_PVMG (blessed 或 magical scalar)</td></tr></table>	SVt_IV	(Integer)	SVt_NV	(Double)	SVt_PV	(String)	SVt_RV	(Reference)	SVt_PVAV	(Array)	SVt_PVHV	(Hash)	SVt_PVCV	(Code)	SVt_PVGV	(Glob)	SVt_PVMG (blessed 或 magical scalar)			
SVt_IV	(Integer)	SVt_NV	(Double)																		
SVt_PV	(String)	SVt_RV	(Reference)																		
SVt_PVAV	(Array)	SVt_PVHV	(Hash)																		
SVt_PVCV	(Code)	SVt_PVGV	(Glob)																		
SVt_PVMG (blessed 或 magical scalar)																					
<code>sv_setref_iv( SV* rv,</code>	创建一个新的 SV, 将其设置为值 I, 并使 rv 指向这个新建 SV。其他的两个函数与之类似。																				

表 20-1 标量变量值的 API (续)

函数 / 宏	描述
<pre>char* classname, int i) (nv 和 pv 与之类似)</pre>	<p>注意 <code>sv_setref_pv</code> 中保存了指针; 它并没有创建一个字符串的拷贝。</p> <p>如果 <code>classname</code> 非空, 这些函数将会将引用 <code>bless</code> 到那个包下。</p>
<code>svREFCNT_dec(SV *)</code>	递减引用记数并在记数为 0 的情况下调用 <code>sv_free</code> 。你自己绝对不能调用 <code>sv_free</code> 。
<pre>SV* sv_bless ( SV *rv, HV* stash); int sv_isa( SV *, char *pkgname); int sv_isobject(SV*);</pre>	<p><code>sv_bless</code> 在一个由 <code>stash</code> 指定的包下 <code>bless</code> <code>rv</code>。请参考“Glob 值与符号表”一节有关 <code>stash</code> 的解释。如果它继承自一个类 <code>pkgname</code> 的话, <code>sv_isa</code> 就会返回 1。</p>
<pre>SV* sv_newmortal() SV* sv_2mortal(SV*) SV* sv_mortalcopy(SV*)</pre>	<p>默认情况下, 如果你创建了一个 <code>SV</code>, 你就有责任将其删除。如果你创建了一个 <i>mortal</i> 或临时变量, Perl 就会自动的在当前作用域结束时将其删除 (除非还有别人保留着对它的引用)。</p> <p><code>sv_2mortal</code> 标记一个现存变量为 <i>mortal</i>, 而 <code>sv_mortalcopy</code> 则创建一个克隆 <i>mortal</i>。</p>
<pre>SV* perl_get_sv( char* varname, int create)</pre>	<p>与你在脚本空间中所常见的一样, 要想获取一个标量变量, 你必须显式的把 <code>SV</code> 绑定到一个名字上。如果 <code>create</code> 为真, 那么在变量以前不存在的情况下就创建它。 <code>varname</code> 必须总是由包名进行限定。例如要创建 <code>\$Foo::a</code>:</p> <pre>SV *s = perl_get_av("Foo::a", 1);</pre>
<code>sv_dump(sv)</code>	名字起的有误, 因为它能够漂亮的打印出所有 Perl 值类型的内容 (在必要时将其转换为 <code>SV*</code> )。这在调试器下运行 Perl 时非常的有用, 比如在 <code>gdb</code> 中, 你可以使用 <code>call sv_dump(sv)</code> 。

表 20-1 中的 *mortal* 系列调用将会创建一个临时的 `SV`, 或是将一个现存值标记为临时的。这些调用本质上告诉 Perl 将 `SV` 推送到一个名为 `tmps_stack` 的堆栈上, 并在当前作用域结束时在 `SV` 上调用 `sv_REFCNT_dec`。(我们将会“其他

堆栈的内幕”一节更多的谈到这个问题。)通常所有在函数之间传递的参数均被标记为 mortal, 因为调用者与被调用的函数都不用操心删除 SV 及其内容的恰当时机; Perl 会自动的处理内存管理问题。

## 这些 API 的应用

也许你的眼睛已经看直了, 而思想呢又多少有些麻木, 使用我们目前见过的 API 来编写一个定制的解释器吧, 让我们减轻一下这种单调乏味的感觉。(目前, 这只是我们的一个有趣的想法!) 例 20-2 描述了一个称做 create\_envt\_vars 的函数, 它为每一个环境变量创建一个标量变量。

例 20-2: 为环境变量创建标量变量 —— 一种比较困难的方式

```
#include <EXTERN.h>
#include <perl.h>
void create_envt_vars (char **environ)
{
    /*
     * environ 中的每个元素都具有 <envt. var name>=<value> 的形式
     */
    SV * sv = NULL;
    char **env = environ; /* 用于对 environ 进行迭代 */
    char buf[1000];        /* 将会包含一个 envt 变量的拷贝 */
    char *envt_var_name;   /* envt 变量名, 就像 PATH */
    char *envt_var_value;  /* 与之相应的值 */
    char var_name[100];    /* 环境变量的全能限定名 */
    while (*env) {
        strcpy (buf, *env);
        /* 检索 "=" 将它替换为 '\0', 于是就可以将它分为
         * 逻辑组成部分 - envt 变量名和值
         */
        envt_var_name = buf; envt_var_value = buf;
        while (*envt_var_value != '=') envt_var_value++;
        *envt_var_value++ = '\0';
        /* 使用包名来限定环境变量
         * PATH 成为 $main::PATH
         */
        strcpy (var_name, "main::"); strcat(var_name, envt_var_name);
        sv = perl_get_sv (var_name, TRUE); /* TRUE => Force Create */
        /* 设置 sv 的字符串值 */
        sv_setpv(sv, envt_var_value);
        env++; /* 接着是下一个环境变量 */
    }
}
```

```

    }
}

static PerlInterpreter *my_perl;
main(int argc, char **argv, char **env) {
    my_perl = perl_alloc();
    perl_construct(my_perl);
    perl_parse(my_perl, NULL, argc, argv, env);
    create_envt_vars(env);
    perl_run(my_perl);
    perl_destruct(my_perl);
    perl_free(my_perl);
}

```

在一台 DEC Alpha 的机器上，你可以像下面这样进行编译和连接：

```

% cc -o ex -I/usr/local/lib/perl5/alpha-dec_osf/5.004/CORE \
-L/usr/local/lib/perl5/alpha-dec_osf/5.004/CORE \
ex.c -lperl -lsocket -lm

```

现在让我们来试一下：

```

% ./ex -e 'print $USER'
sriram

```

太妙了，它成功了——用你普通的 Perl 解释器试一下！好了，其实这并没有什么大不了的，不过你确实在开始着手做一些过去不敢去想或希望去做的事情！

## SV 内幕

一个 SV 有潜力可以变得很大，能够提供在转换成任何一种子类型时所需要的空间。为了避免这种最坏情况的发生，Perl 将信息保留在两个部分当中，如图 20-3 所示，一个通用的称做“sv”的结构用于保存一个位屏蔽标志，一个引用记数以及一个指针 sv\_any，再由它指向一个“特殊部分”。

这个特殊部分是一种相应类型的结构，根据位屏蔽指示的标量变量所包含的内容，它是几种被称做 xpv、xpviv、xpvnv 等结构中的一种。一个标量变量可以刚开始是一个数字，但是在一个字符串的上下文中使用时，它会转变成为一个同时包含数字与字符串两种类型的结构。图 20-3（中间的那个）描述了一个同时包含浮点



数及字符串的SV的例子。假使你使用`sv_setnv`来修改它的值,它就会在`sv_flag`中设置一位来表示字符串部分将不再有效。除非是必要的,否则Perl不会将一个结构进行转换。

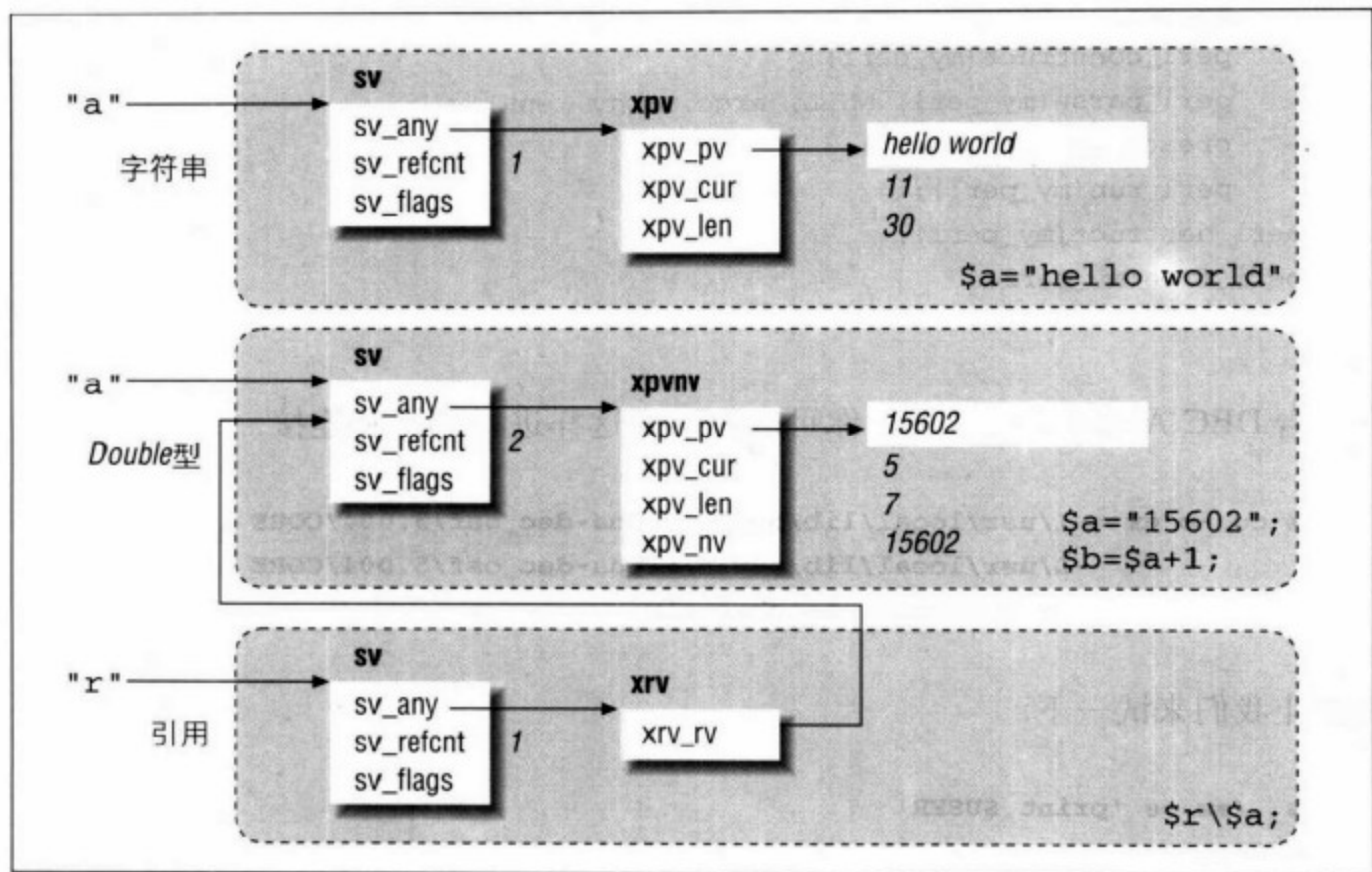


图 20-3 标量变量的内部视图, 每个灰色的方块代表一个 SV

我们前面已经讨论过`Devel::Peek`模块, 可以为你提供脚本一级的对内部信息的存取:

```
use Devel::Peek 'Dump'; # 输入 Dump 子例程
$a = 15602;
Dump ($a);
```

其输出结果为:

```
SV = IV(0x663f1c)
  REFCNT = 1
  FLAGS = (IOK,pIOK)
  IV = 15602
```

将 `$a` 修改为一个字符串来看看会发生什么情况:



```
use Devel::Peek 'Dump';
$a = 10;           # 从一个整数值开始
$a .= "Ten";       # 转换成一个字符串
Dump ($a);
```

其输出结果如下：

```
SV = PVIV(0x7b2ef0)
  REFCNT = 1
  FLAGS = (POK,pPOK)
  IV = 10
  PV = 0x7b2f00 "10 Ten"
  CUR = 6
  LEN = 11
```

请注意，SV 仍然包含有原先的整数值（10），但是由于 FLAGS 字段指示，只有它的字符串内容是有效的，所以那个字段将被忽略。

一个基本的整数值在一个典型的工作站上将花费至少 28 个字节（sizeof(SV) + sizeof(XPVIV) + malloc 的开销）。字符串及数组要比你从长度或字节上推断的更耗费空间。上面的打印结果显示，Perl 为字符串分配了 11 个字节（xpv\_len 字段）而不是最少的 6 个字节（保存在 xpv\_len 字段中的字符串长度）；这意味着你可以至多再添加 5 个字节而不会触发再分配。由于这样做是为了性能与方便考虑的（能够将数字与字符串当作一个实体），所以没有必要如此节省空间。实际上，Perl 将这种以空间换性能的策略应用在几乎所有的数据结构中（注 6）。

没有只包含一个整数或浮点数的如此简单的 xiv 或 xnv。我并不知道这样做的原因，而是胡乱的推测。之所以如此，就是因为一个典型的脚本程序需要将数字转换成字符串（例如，完成 print 操作）或是从字符串转换成数字（读文件）。

图 20-3 中还标明了这三个标量变量的引用记数。中间标量变量的引用记数为 2，这是因为有两个箭头指向它。左边过来的箭头暗示一个符号表条目（全局或局部变量），或我们在第三章所见到的词法变量的便签簿。请注意，所有指向 SV 的指针（实际上是指向任何 Perl 的值）都是指向外层结构的，决不是某些“特殊”部分。

---

注 6： 一个明显的例外就是散列表，它们在一组字符串表中共享键值字符串，这样可以使空间需求最小，但性能上就要逊色一些。

## SV 与对象指针

作为一个扩展编写人员，你时常会对存储由XSUB返回的指向C或C++对象的指针感兴趣。回想一下标量变量的整数分支（IV），保证能够有足够的空间来保存一个指针。我们可以像下面这样来使用这种机制：

```
Matrix *m = new_matrix();  
sv_setiv(sv, (IV) m);           # 将指针转换为一个 IV
```

不管怎么说，这多少有些古怪，但是事实上却能行得通。

在实际应用中，C/C++ 对象通常与经过 bless 的引用相关联，因为它允许 Perl 程序员使用箭头记号（`$matrix->trans-pose()`）。考虑下面的代码：

```
RV *rv = newRV();  
sv_setref_iv(rv, "Matrix", (IV) m);
```

这段代码在内部创建了一个新的整数SV，将它设置为“整数”m，并使rv指向这个新分配的SV。它还在模块Matrix下bless rv。这恰恰像下面你在Perl空间中所表达的那样：

```
my $m = 0xffffa34a;           # 一些指针类型的值， 转换成一个整数  
bless \$m, "Matrix";          # 返回一个经过 bless 的指向 $m 的引用
```

我们将在“使用XS 类型映射来创建对象接口”一节中讨论对象的类型映射时，再来使用这段代码。

## 数组值（AV）

AV是一种动态的包含一组相互毗邻的指向SV指针的数组，正如我们习惯于在脚本空间中所见到的那样。在超出当前容量的索引位置存储一个值，将会触发对该数组进行自动的扩展。表20-2描述了用于整体的操纵数组和对其中单个元素分别进行存取的API。注意，除非你清除或者undef一个数组，否则它不会影响其组成SV的引用记数。

表 20-2 用于操纵数组值的 API

函数 / 宏	描述
AV * newAV() AV * av_make(int num, SV **ptr)	创建一个空的 AV 或是另一个包含 SV* 的数组的克隆。
I32 av_len(AV*);	返回数字索引的最大值 (如 \$#array)。
SV** av_fetch (AV*, I32 index, I32 lval)	从给定的索引中获取 SV*。如果 lval 非零, 它就会将现有值 (在那个位置) 替换成一个 undef。注意 av_fetch 返回的是一个 SV** (而不是 SV*); 这是指向数组中 SV 存储位置的指针。这样你不仅可以修改 SV, 而且还可以对数组本身进行修改 (例如在那个位置拼接数组)。
SV** av_store(AV*, I32 index, SV* val)	在索引位置存储一个 SV* 并像 av_fetch 那样返回一个 SV**。这两个函数都不会更新索引元素的引用记数。
void av_clear (AV*)	递减标量变量成员的引用记数, 并将那些位置替换为 undef。它并不会对数组本身有任何影响。
void av_undef (AV*)	同时减少标量变量成员及数组本身的引用记数。在典型的情况中, 这个函数会释放该数组。与 SV 不同的是, 它们是通过递减引用记数 (SvREFCNT_dec) 而被隐含的删除的。
void av_extend(AV*, int num)	将数组扩展至 num 个元素。尽管其他的函数可以自动的对数组进行扩展, 但是它们只能启发性的决定要扩展多少。如果你需要存储许多的条目, 你可预先扩展数组, 从而节省许多将来可能发生的多次再分配及相应的时间消耗。
void av_push (AV*, SV*)	将一个 SV 从末端推送到 AV 中。如果你想要增添一个列表的话, 你必须还要再编写一些代码。这个与下面的那个函数均不会影响 SV 的引用记数。
SV* av_pop (AV* )	从末端弹出一个 SV, 但并不影响其引用记数, 因此你必须调用 SvREFCNT_dec 或是使用 sv_2mortal 将其标为临时变量, 这样 Perl 将会在作用域结束时将其删除。

表 20-2 用于操纵数组值的 API (续)

函数 / 宏	描述
SV* av_shift(AV*)	与 av_pop 相似，但是它是从 AV 的前端弹出一个 SV。
void av_unshift(AV*, I32 num)	在列表的前端创建 num 个值为空的空间 (将它们填充为 undef)。你必须调用 av_store() 来设置每个元素的值。
AV *perl_get_av ( char* varname, int create)	获取与 varname 相应的 AV。如果 create 为 TRUE 则创建这个变量。

## AV 内幕

与 SV 类似，AV 也被分成通用部分与特殊部分。实际上，对于其他的值类型也是如此。

如图 20-4 所示，字段 xav\_alloc 指向一个动态分配的 SV\* 数组，这是 AV 的实质性内容。av\_fill 中包含了该数组中最后一个有效的 (或填充的) 索引值，而 av\_max 则包含了为数组分配的所有 SV\* 的总数。Perl 总是极力确保分配合理数量的内存空间，这样它就不必每次在向数组中推送一个元素时都要进行 realloc。xav\_array 指向第一个有效元素。它一开始指向 xav\_alloc[0]，然后在 unshift 操作时递增，这样就可以避免不得不将剩余的元素向左移动了。换句话说，AV 的实际内容由 xav\_array 和 av\_fill 进行限定。

指针 xmg\_magic 通常为 NULL，但是如果数组特殊 (如 @ISA)，或代表一个经过 bless 的对象或是与一个包进行了绑定，那么它就会指向一个“魔术”结构 (注 7)。xav\_arylen 是一个 SV\*，它一开始为 NULL，但是当你在数组上使用 \$# 记号时，它就会立刻以魔术变量的形式出现 (用于获取或设置数组的长度)。

Devel::Dump 可以向你提供对数组或其成员标量变量内部细节进行脚本级存取。Dump 要求非标量变量的值要以引用方式传递：

注 7: 在我们讲解魔术变量之前，你无须理解这一段的内容。

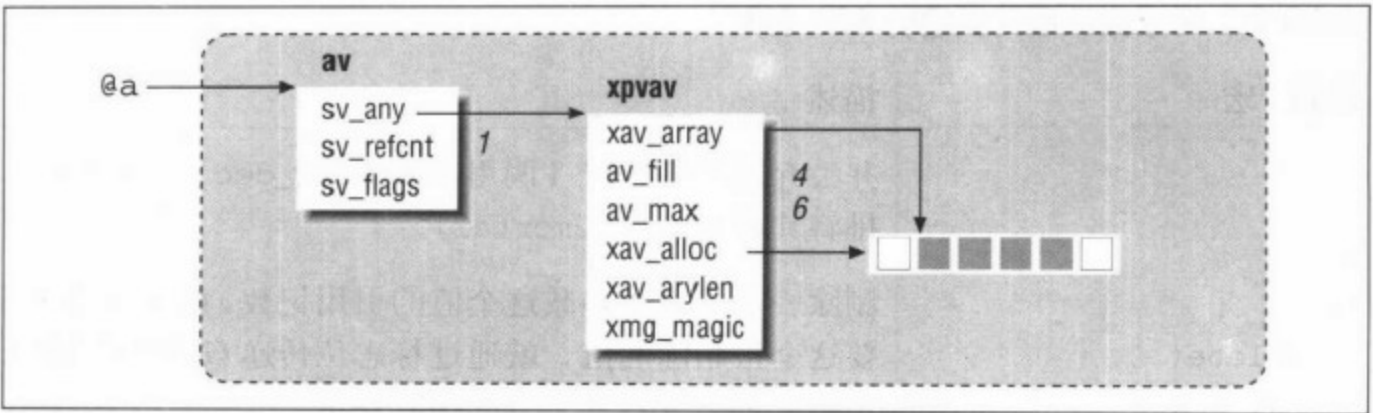


图 20-4 AV 的内部结构，深色的方块中才包含有实际数据

```
use Devel::Dump;
@l = (1,2,3,4);
Dump(\@l); # 以引用方式传递 @l
```

散列表值（HV）

一个 HV 就是包含散列表条目（hash entry，HE）的一张表，其中的每个条目由一对字符串键值和 SV\* 组成。一个散列表中不存在拥有相同键值的散列表条目。表 20-3 中列举的 API 可以允许你将 HV 作为一个整体来操纵，获取或保存单一的元素或是每次一个元素的进行散列表迭代。

表 20-3 用于操纵散列表值的 API

函数 / 宏	描述
HV * newHV()	创建一个散列表值。
SV** hv_store( HV *hash, char* key, U32 klen, SV* val,    U32 hash)	保存键值 - 值对。它并不假定键值为文本字符串，因此你必须提供键值的长度，klen。如果 hash 为 0，那么 Perl 就会自动计算散列表值，这对于普通的 ASCII 键值工作的非常好。与 AV 一样，这些函数并不影响值 val 的引用记数。
SV** hv_fetch( HV *hash, char* key, U32 klen, I32 lval)	和 AV 一样，为了效率而不是方便起见，将返回一个 SV**。在存储一个条目时，解释器将会调用 hv_fetch，来查看是否已经存在了一个包含同样键值的条目。如果有的话，它就能简单的替换该条目的值部分，而不必再次遍历整个结构。通常情况下，你应当将结果间接访问

表 20-3 用于操纵散列表值的 API (续)

函数 / 宏	描述
	并丢弃返回的 SV* (调用 SvREFCNT_dec), 或者安排将其释放 (sv_2mortal)。
SV* hv_delete( HV *hash, char* key, U32 klen, I32 flags)	删除一个条目并递减这个值的引用记数。如果你不再需要这个被删除的值, 就通过标志位传递 G_DISCARD; 不然它就会返回那个值的临时拷贝。 由于那个条目已经从散列表数据结构中删除了, 因此它只需要返回一个 SV*, 而不是 SV**。
void hv_clear(HV *hash)	等价于 %h=()。与 av_clear 类似, 它将保留外层的数组但是要清除散列表条目、键值及值。它还会递减每个值的引用记数 (而不是散列表本身)。
void hv_undef(HV *hash)	清除 HV 并使其引用记数递减。
I32 hv_iterinit(HV *hash)	准备对列表条目进行迭代并返回 HV 中的元素个数。 (译注 2) hv_iterinit 和 hv_itternextsv 由操作符 each, keys 和 values 来使用。
SV* hv_itternextsv( HV *hash, char** key, I32* pkeylen)	获取下一个键值和值。键值以引用的形式返回 (并连同它的长度)。与 hv_fetch 不同, 这个函数只返回一个 SV*。这类似于调用 each()。
HV * perl_get_hv ( char * varname, int create)	获取与 varname 相应的 HV。如果 create 为 TRUE, 那么就创建这个变量。varname 必须由包名进行限定。

迭代函数 (hv\_iter\*) 对于删除操作是安全的, 但是对于插入操作则不是。这就是说, 你可以在使用 hv\_itternextsv 对一个散列表值进行迭代时, 在当前的条目上调用 hv\_delete, 但是你却不能调用 hv\_store, 因为这可能会触发对整个散列表进行重新组织。

译注 2: 此条有误。hv\_iterinit 只返回 xhv\_fill。

## HV 内幕

HV 直接实现了一种称做冲突串联 (collision chaining) 的散列技术。它的基本思想,就是将一个字符串键值缩略为一个整数,并使用这个数字作为进入一个普通动态数组的索引。显然,我们不能期望能将所有的字符串键值缩略为唯一的数组索引,因此这个动态数组的每个元素,都将指向一个包含所有缩略为那个索引值的散列表条目的链表。图 20-5 描述了这种组织。

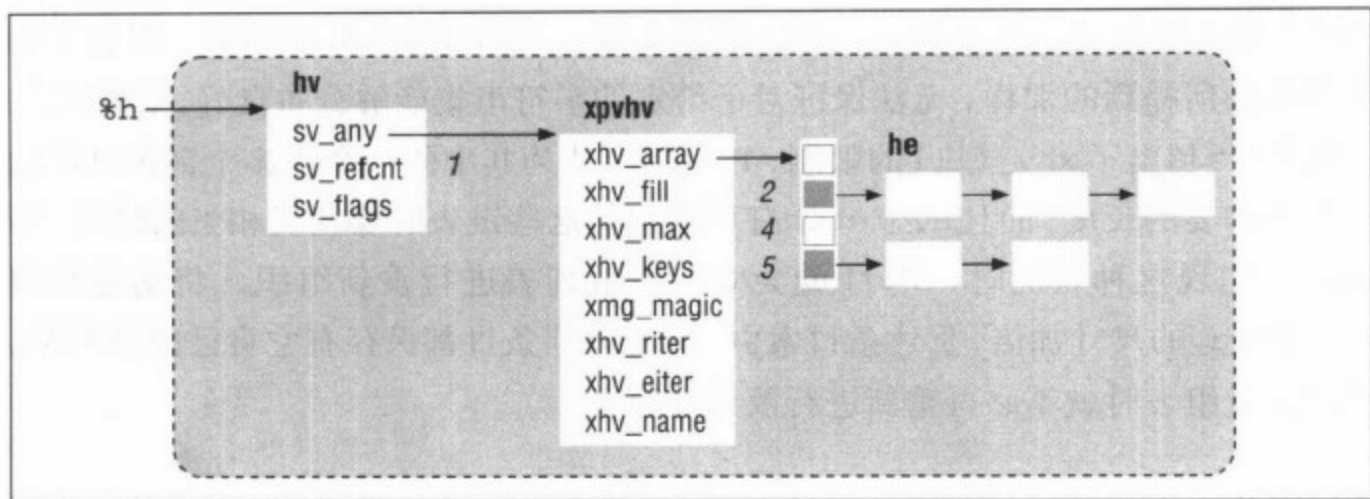


图 20-5 包含散列条目的散列表值

xhv\_array就是我们上面提到的动态数组,xhv\_fill表示允许链表上所悬挂元素的数目,而xhv\_keys包含了散列表条目的总数。给定一个字符串,hv\_fetch()将会计算相应的索引值并遍历对应的链表,并将该键值与每个散列表条目的键值进行比较。

将字符串翻译成一个数组索引需要两步过程(其原因我们马上就要谈到)。首先是以该字符串为参数执行一个称为*hash function*的算法,由它来根据字符串计算出一个整数,而不必担心这个数字是否适合用做一个数组索引。Perl 散列函数的实现如下所示:

```

int i = klen;
unsigned int hash = 0;
char *s = key;
while (i--)
    hash = hash * 33 + *s++;

```



结果所产生的数字就称做一个散列。无法保证不同的字符串将产生不同的散列值。注意，如果你有一种不同的散列算法，你可以自己计算散列值并将其提供给 `hv_store`（请参考表 20-3）。为了将散列值转换成一个实用的数组索引，Perl 将其与数组的最大尺寸值进行混合：

```
index = hash & xhv_max;
```

在理想情况下，我希望这些条目能够均匀的散布在数组中，以保持链表的短小。Perl 的散列算法对于通常的 ASCII 字符串来说，工作的令人吃惊的好，但是正如我们前面所提到的那样，无法保证对于给定的字符串集能够分布良好。因此如果 `xhv_keys` 超出了 `xhv_fill` 的限制，Perl 就会认为其中的一个或多个链表已经达到了不必要的长度，而且 `hv_fetch` 有可能遍历这些链表时要花费相当的时间。所以，当出现这种状况时，Perl 就会立即对散列表进行重新组织：将动态数组 `xhv_array` 的尺寸加倍，重建条目索引。每个散列条目都保存有它自己的散列值，因此在重组表时就不必再重新进行散列计算了。

你可以像下面这样通过在标量变量上下文中打印出一个关联数组，来了解一下散列表的效率：

```
# 创建一个散列表
for (1 .. 1000) {$h{'foo' . $_} = 1;} # 创建 1000 个条目
print scalar(%h);
```

在我的机器上其输出结果为“406/1024”，这只是 `xhv_fill` 与 `xhv_max` 之间的比值。比值越低，对散列表的存取就越快，因为这样链表的平均长度就短些。

如果你自己要在一个散列表上进行大量的插入操作的话，你可以像下面这样让它预先分配特定数量的动态数组，来改善脚本空间中的运行效率：

```
keys %h = 400; # 设置 xhv_max
```

Perl 会将其累加成下一个比它大的 2 的幂：512。

字段 `xhv_iter` 与 `xhv_either` 由迭代函数 `hv_iterinit` 和 `hv_iteernextsv` 使用，并构建一个散列条目上的游标。`xhv_riter` 包含了当前的行索引而 `xhv_either` 包含了指向当前条目的指针。



大多数面向对象的Perl实现都利用散列表来存储对象属性，也就意味着一个给定类的实例通常会拥有相同的键值字符串集合。为了防止不必要的重复，实际的键值字符串放在一个系统范围的共享字符串表中进行维护（*strtab.h*中的*strtab*）。*strtab*是一种简化的HV：这里的每一个值保留一个对那个字符串使用数量的引用记数。当遇到“*\$h{'foo'}*”时，字符串*foo*将会第一次进入*strtab*中，前提是以前并不存在。然后*\$h{foo}*的散列表条目将在*%h*的HV中创建。结果性能所受的影响非常小；如果存在大量重复键值的话，这种共享存储就能够节约不少时间，因为键值只被*malloc*一次。同样，由于散列算法也只需执行一遍，所以即便没有那么多重复性能也是相当的好。

这种共享字符串表只能应用于固定不变的字符串（要知道散列表的键字符串是不能够改变的）。包含用户定义字符串的SV是不能使用这张表的。

## Glob 值与符号表

我们在第三章已经见过了*typeglob*，也被称做*glob*值或GV，将其他的值类型与一个符号表条目进行连接的情况。一个标识符名如“*foo*”通过GV与*\$foo*、*@foo*、*%foo*，名为*foo*的文件句柄以及名为*foo*的打印格式进行连接。

GV与符号表如此密切的一起工作，以至于所有操纵符号表的代码也整个被放置在*gv.c*中了。符号表在内部是以散列表（HV）方式实现的，因此也被称做*stash*（符号表散列的缩写）。每个包都有其自己的*stash*并且还包含有指向嵌套包的*stash*的指针。可以从一个名为*defstash*的全局变量（注8）中获得的主*stash*中包含了指向其他“顶层”包*stash*的指针。表20-4描述了用于存取GV及符号表的重要函数。

表20-4 用于操纵Glob值及Stash的API

函数 / 宏	描述
GvSV, GvAV, GvHV, GvIO, GvFORM	返回悬挂在GV上的相应类型的值指针。

注8： 如果定义了MULTIPLICITY的话，就是每个解释器有一个这样的变量。

表 20-4 用于操纵 Glob 值及 Stash 的 API (续)

函数 / 宏	描述
HV *gv_stashpv( char *name, int create)	给定一个包名, 获取相应的 HV。名字后面的无须有 "::", 这与脚本空间中的情况不同。
HV *gv_stashsv( SV *, int create)	与上面的相同。SV* 包含有包的名字。
HV *SvSTASH (SV* sv)	从一个经过 bless 的对象获取 stash。如果 sv 是一个引用, 则首先要对它进行间接访问: SvSTASH(SvRV(sv))。
char* HvNAME(HV* stash)	给定一个 stash, 返回包的名字。

脚本空间中的标准变量如 \$\_, \$@, \$&, \$` 和 \$' 在 C 空间中以全局变量来使用, 即分别对应于 defgv、errgv、amperv、leftgv 和 rightgv。例如, 如果你知道 \$\_ 中包含有一个数字, 那么你就可以像下面这样将其提取出来:

```
int i = SvIV(GvSV(defgv)); /* $_ 与 @_ 由 defgv 表示 */
```

## Glob 值与符号表内幕

图 20-6 中描述了大多数 GV 中有趣的组成部分。

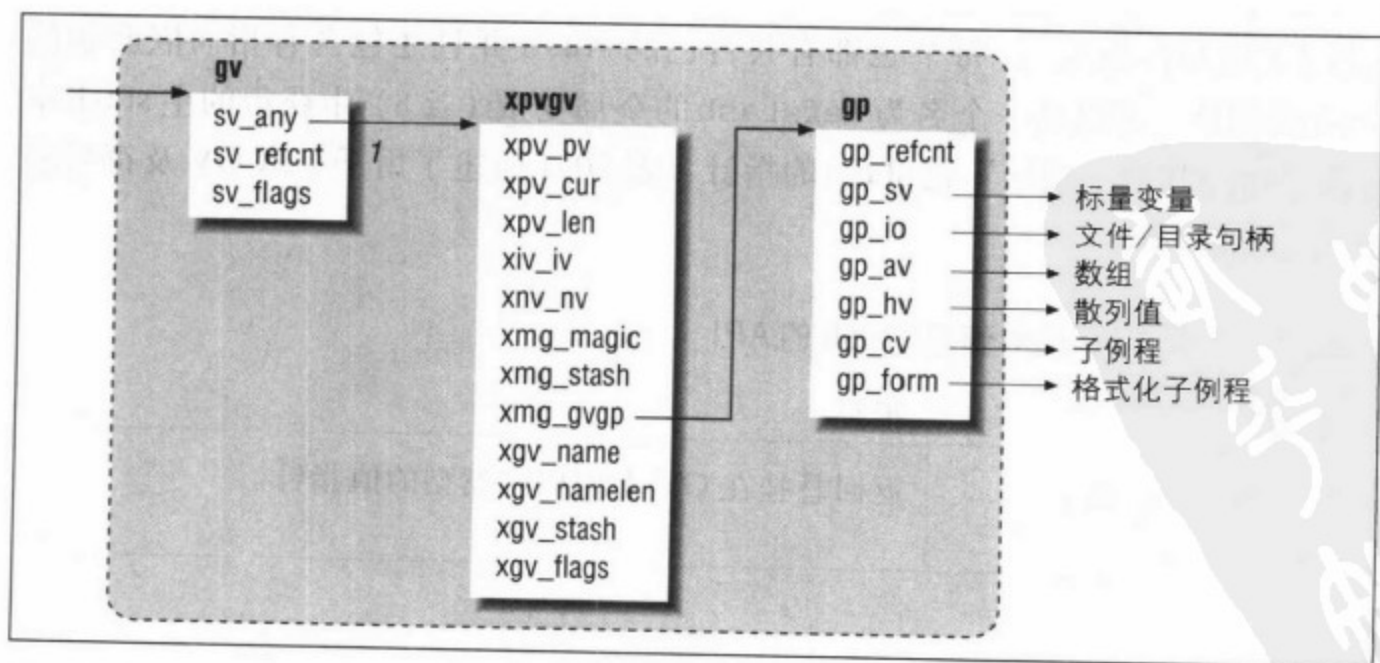


图 20-6 Glob 值结构

xgv\_name 字段保存了变量名（不带前缀）。指向被包含值（\$foo、@foo 等等）的指针单独封装在一个称做 gp 的结构中，以进行快速的别名处理。在通常不同的变量类型使用不同名称的情况下，所有值指针除一个外全为 NULL。

符号表为 HV，由它将变量名映射到 GV 中。但是难道 HV 只能存储 SV 吗？其实，你或许已经注意到了，所有的值类型外观相同的包裹结构，由它们来维护诸如引用记数、标志位以及指向内部结构的指针等信息。因为它们完全相同，因此你将一个 AV\*、HV\*、CV\* 转换成一个 SV\*，也就能欺骗 HV 来存储任何你需要的东西。如果你自己要来这么做的话，一定要注意那些递减所包含“SV”引用记数的 HV 调用（如 hv\_delete）。这是因为在引用记数变为 0 时会触发一个 sv\_free 操作，而如果它不是 SV 的话，你就会有麻烦了。

## 代码值

我们现在已经学完了在 Perl 中使用的所有基础数据类型。下一步我们将学习代码值，它代表子例程、eval 代码块和打印格式声明。这些解释将能够使你从 C 中高效的调用 Perl 子例程，并且还可以使你深刻的理解词法变量及闭包是如何实现的。

表 20-5 中描述了用于操纵 CV 的 API；对 CV 来说你除了调用它以外可做的事情并不多。除了 perl\_call\_sv，所有其他的 API 函数接受的均为过程名而不是 CV 本身。

表 20-5 用于操纵代码值的 API

函数 / 宏	描述
CV* perl_get_cv(char *name, int create)	获取给定名字的 CV。你应当总是将 create 设置为 FALSE，因为 TRUE 值将会在子例程不存在时自动创建一个空白的 CV，这对于应用编写人员来说没有任何意义。
int perl_call_sv(SV* cv, int flags)	调用由参数 cv 指示的子例程。（不错，你需将 CV* 转换成一个 SV*。）它将返回压入到堆栈中的返回值的个数。flags 的意义下面解释。

表 20-5 用于操纵代码值的 API (续)

函数 / 宏	描述
<pre>perl_call_argv(     char *sub,         I32 flags,         char **argv);</pre>	见表 19-1 中的内容。
<pre>perl_call_va (     char *sub,         [char *type, arg], *         ["OUT", ]         [char *type, arg,]*     );</pre>	我们已在表 19-1 中解释过了。我们将在后面的“简单的嵌入式 API”一节中来实现这个函数。
<pre>int perl_call_pv (     char* sub_name,     int flags)</pre>	通过名字调用子例程。它是建立在 perl_call_sv 上的一层薄薄的包裹层。
<pre>int perl_call_method(     char *method_name,     int flags)</pre>	通过名字来调用一个类的方法。堆栈上的第一个参数必须要么是包含类名的 SV 要么就是经过 bless 的类的引用。

还有其他调用 Perl 子例程的方法，如 perl\_call\_argv 和 perl\_call\_va，这些我们已经在上一章中看到过了。所有这些函数均为 perl\_call\_sv 的包裹函数，并力图在一定程度上隐藏消息传递协议的细节。参数 *flags* 为下面定义在 *perl.h* 中位屏蔽的组合：

#### G\_DISCARD

忽略函数的所有返回参数

#### G\_SCALAR, G\_ARRAY

指定一个标量变量或数组上下文，默认为标量变量。被调用的子例程可以通过 wantarray 来查明调用者的目的。这些标志还可以同 G\_DISCARD 结合在一起使用。这在你想影响调用 wantarray 的函数的工作方式时是很有用，即便是你对结果不感兴趣。

## G\_EVAL, G\_KEEPER

用调用来包裹一个 eval 代码块。perl\_eval\_sv() 将会自动假定设置了该标志位。在一个经过 eval 的代码块终止时, Perl 将会将 die 的字符串参数赋值为 errgv(\$@), 并将所有在那个代码块中创建的临时变量清除掉。Perl 将检查这些变量中是否存在一个经过 bless 的对象, 如果有, 它就会调用其 DESTROY 例程。存在这种例程可以调用 die 的可能性 (毕竟它是由用户定义的代码)。这时就会出现这种情况, errgv 已经计算得出了结果而抛出了一个附加的例外。通过使用 G\_KEEPER, 你将指示 Perl 将这个新的例外字符串与原先的 errgv 进行连接, 而不是将其覆盖。

## CV 内幕

CV 整体上的结构与其他值类型相同: 一个通用的部分以及一个特殊部分。考虑下面的代码, 它在另一个包中定义了一个函数 (通过对名字进行完全限定), 并使用 Devel::Peek 对该函数进行检查:

```
package Foo;
sub main::bar {    # 将函数引入一个不同的包中
    my $a = 10;
}
use Devel::Peek;
Dump(\&main::bar);
```

打印输出的大概结果如下:

```
SV = PVCV(0x774300)
  REFCNT = 2
  FLAGS = ()
  IV = 0
  NV = 0
  COMP_STASH = 0x6635f0 "Foo"
  START = 0x7744d0
  ROOT = 0x774650
  XSUB = 0x0
  XSUBANY = 0
  GVG::GV = 0x66365c "main" :: "bar"
  FILEGV = 0x660418 "_<foo.pl"
  DEPTH = 0
  PADLIST = 0x66362c
```



字段 `COMP_STASH` 表示在 `bar()` 执行时, “Foo” stash 将会是活动的, 尽管该子例程是在包 `main` 中定义的。 `ROOT` 字段表示 CV 语法树的根操作码, 而 `START` 就是在函数开始执行时要获取控制的操作码的地址。 `XSUB` 字段要么为 `NULL`, 要么包含有一个指向 C 子例程的指针。 `DEPTH` 字段表示递归的深度, 而 `PADLIST` 指向一组用于存储子例程中定义的词法变量的便签簿。下一节我们将更多的谈到这个问题。

### local 与 my 是如何工作的

我们都知道, Perl 变量可以是全局的、动态的(以 `local` 进行标注)或词法的(`my`)。全局变量通过 stash 和相应的 `typeglob` 进行存取。当 Perl 遇到全局变量 `$a` 时, 它就会生成 `gvsv` 操作码, 由它在运行时将相应 GV 的标量变量放置到堆栈上。

当 Perl 对 “`local $a`” 进行语法分析时, 它会产生同样的操作码 `gvsv`, 但是这一次会在那个操作码中设置一个特殊的标志位, 以将标量变量“局部化”。在运行时, 相应的操作码函数 `pp_gvsv` 会检查这个标志位, 如果被设置了, 则将 GV 的标量变量值替换为一个新的标量变量值, 并将这个新值推送到参数栈上。同时, 原先的 SV 将被安全的保留在一种称做存储堆栈 (save strack) 的设施中(我们将在后面“其他堆栈的内幕”一节中再进行讨论)。后续的对那个作用域(或嵌套作用域)中 `$a` 的存取, 将通过 `a` 的 GV 导向一个新分配的标量变量值。

对 `my` 变量的存储及处理则大相径庭。我们以前提到过, 每个 CV 都包含有一个 `padlist`, 如图 20-7 所示, 这是一组便签簿的列表。

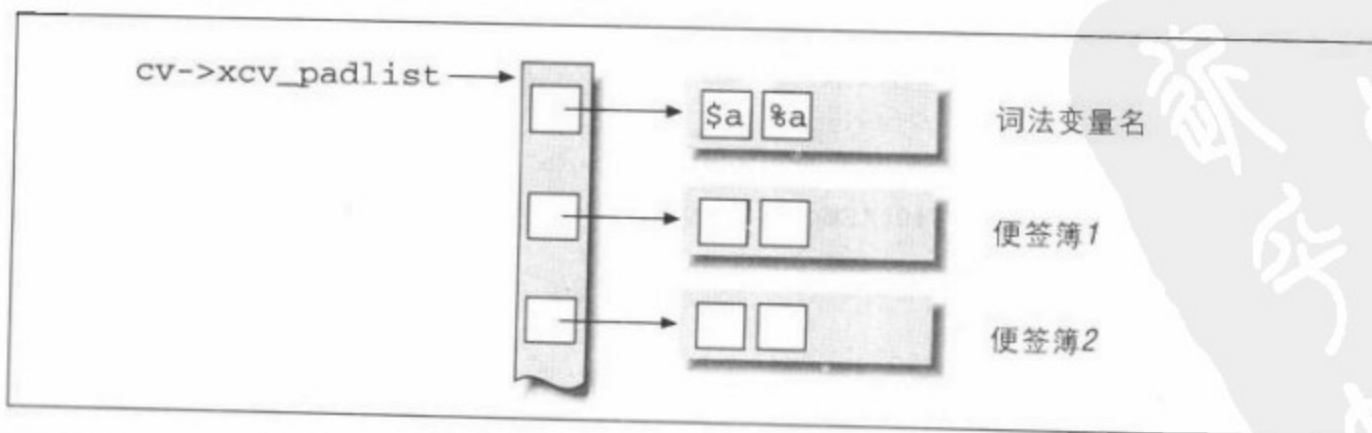


图 20-7 my 变量的内部视图

这个 `padlist` 是一个普通的 AV。它的第 0 个元素指向一个包含在那个子例程中使用的（并不仅仅是那些进行声明的）所有词法变量名的 AV。这些名字包含了其前缀符号，因此 `$a` 与 `%a` 将拥有不同的条目。`padlist` 的第一个元素指向一个便签簿数组（也是一个 AV），其中的元素包含了对应于第零行中词法变量名的值。如你所见，这个 `padlist` 是另一种形式的符号表，因为它中间也包含了逻辑上成对的变量名和值。

当子例程进行递归时，将会为每一层递归分配一个新的便签簿（注 9）。你将会注意到一个 CV 至少需要三个 AV（一个是为了 `xcv_padlist`，一个用于存储名字，而其他或更多的则用来存储值）。

在多线程引入到 Perl 中时（5.005 以上的版本），每个线程将拥有它自己的便签簿，也就意味着词法变量对于一个递归层次及线程，将继续保持完全的私有性（包的全局变量当然仍保留其全局性）。

`my` 变量比起 `local` 变量要稍快一些。原因就是 `local` 在运行时要分配一个新值来临时屏蔽全局值。而 `my` 变量对于 CV 来说已经是唯一的了，因此它们通常在进行语法分析时只分配一次。唯一需要创建一个崭新的 `my` 值的理由，就是碰到递归时的情况，但这并不典型。在 Perl 未来的发行版中，执行同一个 CV 的多个线程也将需要运行时词法变量的分配。

当你存取词法变量时，Perl 的代码生成器就会输出一个称做 `padsv` 的操作码，它等同于 `gvsv`（它应用于全局或局部变量）。`padsv` 将会记住变量在便签簿中的偏移量（对于图 20-7 中的 `%a` 就是 1）。在运行时，Perl 根本不费任何时间就能够取得相应的值并将其推送到堆栈上。

## 闭包

简单介绍了 CV 和词法变量之后，再来引出闭包的主题。当创建一个闭包时，Perl 分配一个 CV，使之指向那个子例程的起始操作码，并向其提供属于它自己的私

---

注 9： 这是根据 Malcolm Beattie 的当前支持 POSIX 线程的原型补丁程序来说的。



有 padlist。这个 padlist 包含了指向所有由那个闭包使用的词法变量的指针，而无论它们是否是在那个代码块中创建的。如图 20-8 所示。

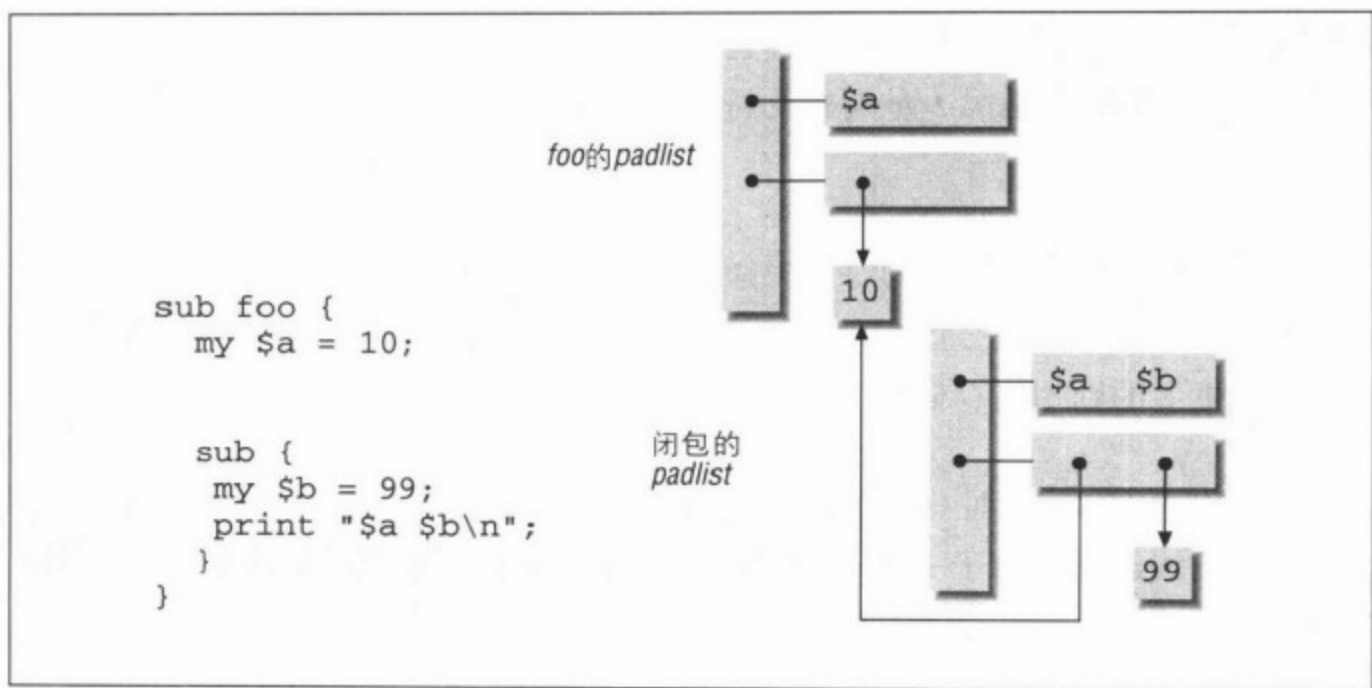


图 20-8 从包含其 CV 的便签簿中提取词法变量值的便签簿

对于那些从包含闭包的 CV 中提取的词法变量值来说，便签簿中包含了直接指向相应值的指针，而这些值的引用记数也被递增。诸如 \$b 这样的在闭包中创建的词法变量是刚刚才分配的。无论是哪一种情况，对于为一个词法变量而分配的存储空间来说，只要存在可以调用的使用该变量的子例程，它就不会被释放。

## 对象与闭包的对比

在第四章“子例程引用及闭包”中，我们已经注意到了对象与闭包之间的相似性：它们两个都表示代码与数据之间的一种绑定。换句话说，一个包含三个成员函数的对象可以由三个闭包来表示，而它们又作用于从包含它们的环境中转借的私有变量。

显然，采用闭包的方式空间消耗要大得多；要表示 100 个对象所代表的信息，你就需要 300 个唯一的闭包，也就是需要 900 个 AV。与之对照，如果你使用散列表的表示形式来存储对象属性，则只需要 100 个散列表和 9 个 AV（每个子例程 3 个）。



另一方面，调用一个闭包要比调用一个对象的方法快一些。这是因为一旦过程被调用闭包的变量就可以被立即使用，而对象的方法则必须先对对象引用进行间接访问，然后对每个属性还要再进行散列表存取。下面的性能测试比较了对象的存取方法与同等闭包之间的速度差异——在我的机器上后者的速度要快两三倍：

```
# -----
package OBJECT;                                # timing 对象存取函数的包
sub new {
    bless {'abc' => 10};
}
sub abc {                                       # 获取属性 abc
    $_[0]->{'abc'};
}
sub increment {                                # 递增属性 abc
    $_[0]->{'abc'}++;
}
#-----
package CLOSURE;                                # timing 闭包的包
sub new {
    my $abc = 10;                             # 成员数据
    my($rs_increment,$rs_abc);
    $rs_increment = sub {$abc++};             # 等价于 OBJECT::increment
    $rs_abc        = sub {$abc} ;             # 等价于 OBJECT::abc
    ($rs_increment, $rs_abc);
}
#-----
package main;
use Benchmark;
$a = OBJECT->new();                            # 新建一个对象
($inc, $fetch) = CLOSURE->new();               # 创建两个闭包
timethese(1000000, {
    Object => '$a->increment',                 # 调用一个对象的方法
    Closure => '&$inc'                         # 调用一个闭包
});
```

在我的 PC 上，打印结果为：

```
Benchmark: timing 1000000 iterations of Closure, Object...
Closure: 13 secs (14.39 usr 0.00 sys = 14.39 cpu)
Object: 45 secs (45.14 usr 0.00 sys = 45.14 cpu)
```

## 魔术变量（注 10）

存在用户定义的包含字符串、数字及引用的普通变量；同时还有魔术（magical）变量，这些变量都有一个或多个特殊属性。例如一个绑定的变量就是一个魔术变量，因为它包含了指向绑定对象的指针，并在对其进行读写或写操作时调用对象的 FETCH 和 STORE 方法。这一点我们已经在第九章“绑定”中见到过了。那些诸如 \$! 和 %SIG 的内建变量也同样特殊：在读取 \$! 时，Perl 将隐含的读取 C 变量 errno；当向 %SIG 写入时，Perl 就会复位信号句柄。

图 20-9 描述了一个魔术标量变量。它包含了你以前所看到的普通的标量变量字段，而且还指向了一个属性链表。由一个名为 MAGIC 的结构来表示每个属性，而且还为不同类型的属性提供一致性的伪装，这一点我们很快就会看到。让我们利用这种机制之前，先来详细的看一下这个结构。

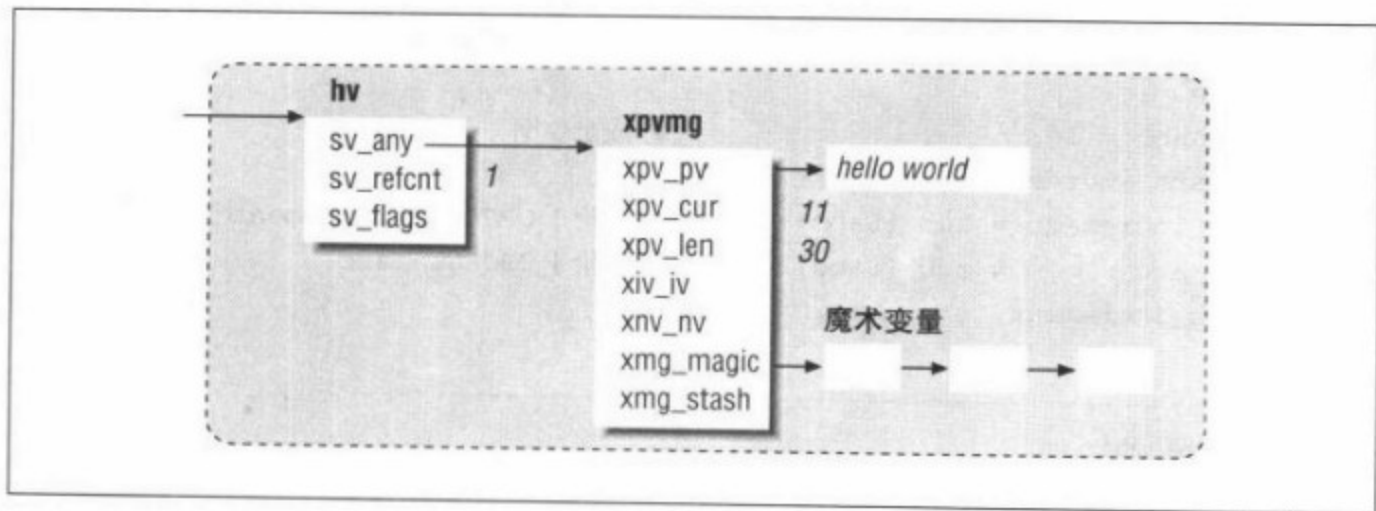


图 20-9 魔术标量变量

如图 20-10 所示，一个属性就是一个对象，它包含了一个属性类型，一个指向一些这个属性私有数据的指针以及一个指向虚表（或 *vtbl*，用 C++ 的话来说就是一张包含指向函数的指针的表）的指针。当对一个变量进行读取、写入、清除、销毁或在存取长度时，Perl 首先更新变量的值（字符串，整数或浮点数字段），然后调用对相应操作（读、写、清除，等等；请参考图 20-10）负责的存取函数。如果变量的属性多于一个，那么每个与属性相应的存取函数均被调用，以使它们都

注 10： 在首次阅读时可以跳过这一节。

有机会任意的影响变量的值。存取函数还可以有副作用。例如，当你修改 %SIG 时，它的属性的每个 svt\_set 函数均被调用。这些函数的其中一个用于更新信号处理句柄。

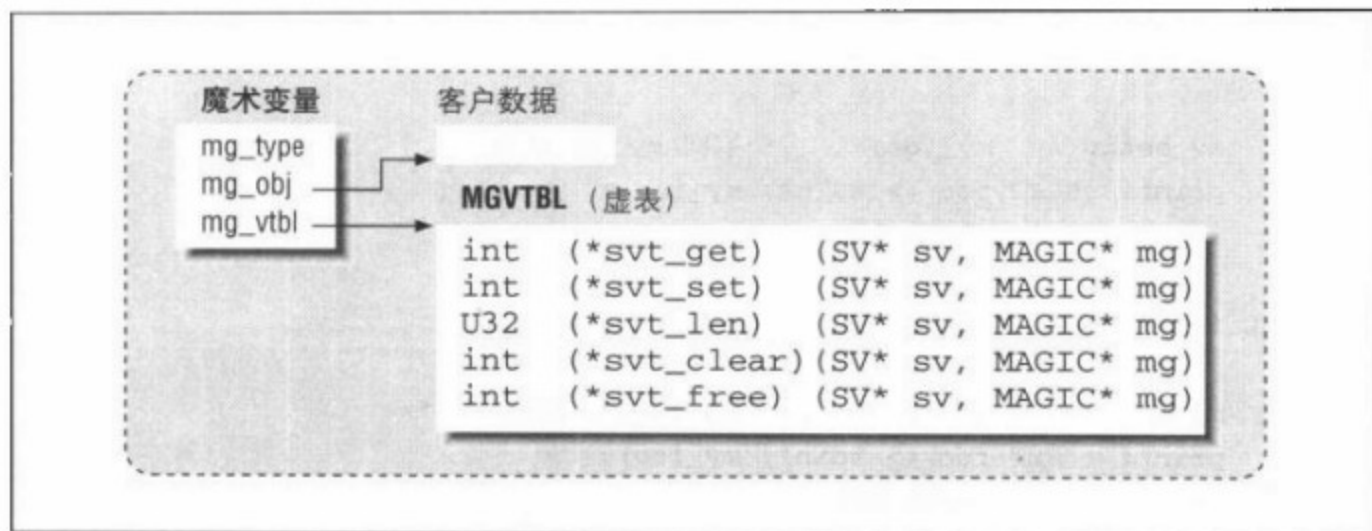


图 20-10 魔术变量：用于表示特殊属性的统一接口

Perl 带有一组与唯一属性类型相关的预先建立的虚表，它们只是一些唯一的属性特征。例如，用于处理绑定数组的虚表由字符“P.”来表示。如果你还对其他的内建类型感兴趣的话，可以参考一下 *perl guts* 文档。一个值只能拥有至多一个列表中的给定类型的属性。有一个由字符 ~ 标识的属性类型，它是一个让扩展编写人员提供定制虚表的钩子。我们来看一下如何来使用这个特殊的类型。

要想为一个标量变量安装特殊的属性，可以像下面这样使用函数 `sv_magic`:

```
sv_magic(sv, obj, '~', "foo", 3);
```

该函数在内部将标量变量值升级为一个 XPVMG 结构，并告诉 Perl 不要安装任何它所预先定义的虚表（这是因为 ~）。此外它还创建了一个 MAGIC 结构，并将其挂在标量变量上。obj 是由你选择的 SV，它包含了用户定义的数据，并为了让存取函数来区分各个魔术变量。最后的两个参数只是为属性起了个名字：一个标识字符串以及其长度信息。通常来说你使用变量的名字。

你可以使用函数 `mg_find` 从标量变量中存取特定的属性结构：

```
MAGIC *m = mg_find(sv, '~');
```

让我们使用这两个函数来创建一种底层的绑定机制：在对变量进行某种操作时调用一个定制的函数。下面例子中的过程 `foo_tie` 描述了如何将一个 Perl 空间中的变量 `$foo` 与一个 C 变量关联起来：

```
int my_foo; /* 在脚本一级绑定到 $foo */
int foo_get (SV *sv, MAGIC *mg)
{
    sv_setiv(sv, my_foo); /* 返回 my_foo 的值 */
    printf ("GET foo => %d\n", my_foo);
    return 1; /* 返回值无用 */
}
int foo_set (SV *sv, MAGIC *mg)
{
    my_foo = SvIV(sv); /* 设置 my_foo 的值 */
    printf ("SET foo => %d\n", my_foo);
    return 1; /* 返回值无用 */
}
MGVTBL foo_accessors = { /* 定制虚表 */
    foo_get, foo_set, NULL, NULL, NULL
};
void foo_tie ()
{
    MAGIC *m;
    /* 创建一个变量 */
    char *var = "main::foo";
    SV *sv = perl_get_sv(var, TRUE);
    /* 将 sv 升级为一个魔术变量 */
    sv_magic(sv, NULL, '~', var, strlen(var));
    /* sv_magic 为 sv 增加一个 MAGIC 结构 (类型为 '~')。
       获取并设置虚表指针 */
    m = mg_find(sv, '~');
    m->mg_virtual = &foo_accessors;
    SvMAGICAL_on(sv);
}
```

由于 `foo_tie` 使用了 '~' 属性类型，因此 Perl 将不提供预先建立的虚表。`foo_tie` 通过提供它自己定制的虚表 `foo_accessors` 来弥补这个空缺。这个虚表中包含了指向 `foo_get` 和 `foo_set` 的指针。请注意，这两个函数存取的是给定标量变量的整数域。

在脚本级可用的 `tie` 机制要稍微更复杂一些。它首先要求模块返回一个对象（使用 `TIESCALAR`，`TIEHASH` 等等），并将这个对象作为 `sv_magic` 的参数。以后每当对标量变量进行读取时，就会调用 `sv_get`，而它又将调用转嫁给私有对象的 `FETCH` 方法。

## 堆栈与消息传递协议

好了，我们现在已经非常深刻的了解了由 Perl 所提供的所有值类型的情况。下面的半章将用来理解调用者与被调用子例程之间所使用的数据结构、API 和协议。

我们前面就提到过，参数堆栈是用于在函数之间传递参数和结果的数据结构。图 20-11 描述了调用 `foo(10,20)`，而它又依次调用 `bar("hello", 30.2, 100)` 时的堆栈情况。

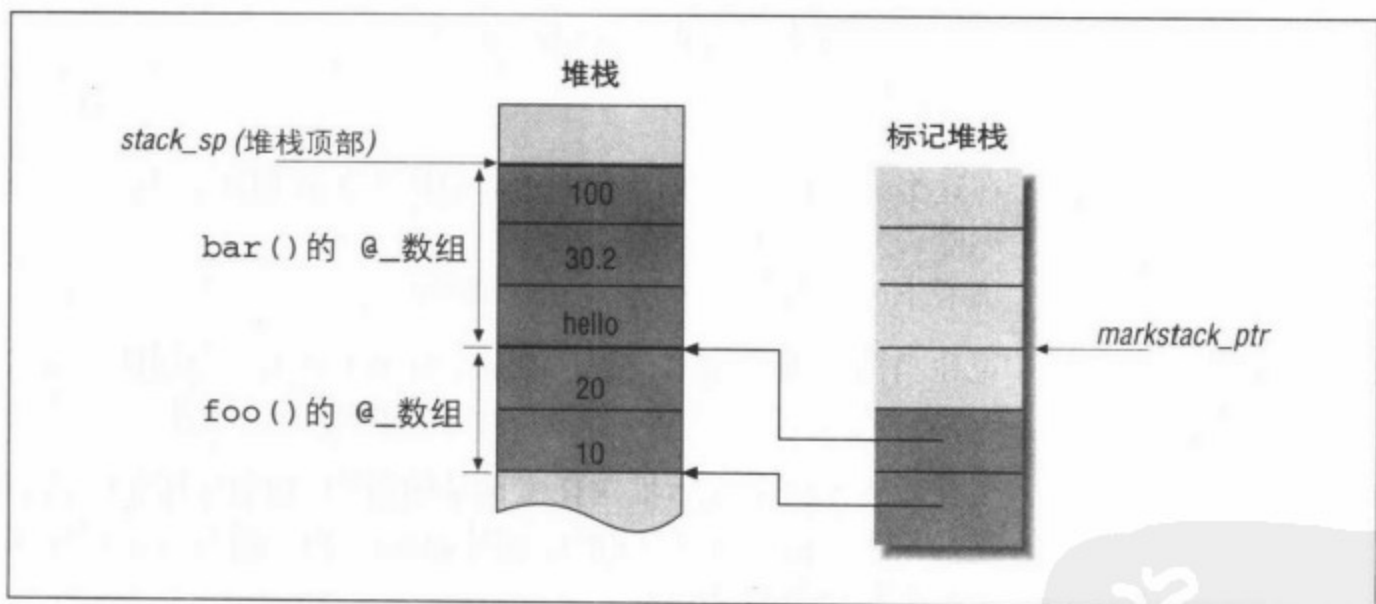


图 20-11 在 `foo` 调用了 `bar` 而 `foo` 又被调用时的参数及标记堆栈情况

`bar` 是如何知道它要从堆栈顶部提取多少个参数的呢？其实，Perl 通过另一个名为 `markstack` 的标记堆栈（某种意义上讲就是一组书签）来跟踪保留参数堆栈的跨度。`bar` 通过简单的计算当前堆栈顶部与存储在 `markstack` 顶部的书签之间的差值，就会知道那些参数是给它的了。这一段堆栈数据对应于 `bar` 的 `@_` 数组。反过来，当 `bar` 将要返回时，它就会将一个或多个结果卸载到它的那段堆栈中，而 `foo` 则可以通过查看 `markstack` 来获知究竟返回了多少标量变量。

当你在脚本空间中时所有这些操作都是透明的。但是如果你编写由 Perl 调用的 C 例程（扩展 Perl）或者从 C 中调用 Perl 函数（嵌入 Perl）时，就需要处理一些细节问题。你会发现后面的章节将为你编写更为强大和易于理解的扩展铺平道路（这里的易于理解是对脚本编写人员来说的）。

## 调用一个 Perl 子例程

让我们先从 C 中调用 Perl 子例程的情况开始，这通常应用在把 Perl 解释器嵌入到你的应用中的时候。表 20-6 包含了你需要使用的宏的描述，你要按照给定的顺序来进行使用。这些宏乍一看来或许难以记忆，但是幸运的是它们每次都以完全相同的顺序来调用，而你使用一段时间就会适应的。

表 20-6 在调用 Perl 例程时使用的宏（解释器的嵌入）

函数 / 宏	描述
DSP	声明由下面的宏来使用的变量。
ENTER	起始作用域。
SAVETMPS	在这个调用后创建的所有 mortal 变量均在调用 FREETMPS 时被删除。 请看下一节有关 tmps_stack 的解释。
PUSHMARK	记住当前堆栈的顶部位置（更新 markstack）。调用 ENTER, SAVETMPS 和 PUSHMARK 来为一个子例程调用做准备。
XPUSHs (SV*)	现在你可以向堆栈中推送任意数量的参数。如果你推送的是新建的 SV，那么可以将它们标记为 mortal 的，而 Perl 会在作用域结束时将其自动删除。
PUTBACK	表示所有的参数均被推送完毕。PUSHMARK 和 PUTBACK 在某种意义上将是将参数括起来的。这时，Perl 过程可以通过使用 perl_call_pv 或 perl_call_sv 来调用。（请看下面的例子。）
SPAGAIN	与 PUSHMARK 相似，它为返回结果提供开放的括号。即便是没有任何返回结果，你也必须要调用它。

表 20-6 在调用 Perl 例程时使用的宏（解释器的嵌入）（续）

函数 / 宏	描述
POPi	从堆栈中弹出一个标量变量并返回相应的类型：整数，长整数，双精度浮点数，指针（通常指向字符串）以及 SV。 perl_call_pv 返回推送到堆栈上的返回参数的个数，而你必须在多次调用这些宏时倍加小心。
POP1	
POPn	
POPp	
POPs	
PUTBACK	要时刻记住 POP 以与 Perl 过程向栈中推送结果相反的顺序来返回结果。
PUTBACK	当所有的参数均被弹出以后调用它。
FREETMPS	请查看 SAVETMPS。
LEAVE	结束作用域。请查看 ENTER。

接下来的代码片段，描述了如何以两个输入参数 10 和 20 调用名为 add 的 Perl 过程，以及如何获取返回结果。再次提醒大家要按照表 20-6 中的顺序来使用这些宏。

```
#include <perl.h>
void foo() {
    int n;          /* 由 add 返回的参数个数 */
    dSP;
    ENTER;          /* 告诉 Perl 现在进入了一个新的作用域 */
    SAVETMPS;        /* 确保 FREETMPS 只释放那些在 stmt
                       之后创建的 mortals */
    PUSHMARK(sp);    /* 记住当前的堆栈指针 .sp 由 dSP 声明 */
    /* Push arguments */
    XPUSHs(sv_2mortal(newSViv(10))); /* 推送一个整数 */
    XPUSHs(sv_2mortal(newSViv(20))); /* 再推送一个 */
    PUTBACK;         /* 参数结束 */

    /* 使用名字来调用子例程，并期望它返回一个标量变量 */
    n = perl_call_pv("add", G_SCALAR);

    SPAGAIN;         /* 开始查看返回结果 */

    /* 从堆栈中获取返回值 */
    if (n == 1)
        printf("Result: %d \n", POPi);
    /* Closing details */
    PUTBACK;         /* 完成从堆栈中删除返回值的操作 */
}
```



```

    /* Time to clean up and leave .. */
    FREEMPS; /* 释放这两个传递给 add 的临时变量 */
    LEAVE;    /* 离开作用域 */
}

```

这就是你所需要的理解“简单的嵌入式 API”一节的内容的所有信息，那一节实现了我们在第十九章引入的便利函数 `perl_call_va`。

## 被调用方：手工编制 XSUB

在了解了调用 Perl 子例程所需要的知识之后，让我们再来从一个被调用子例程的角度看一下堆栈的情况。这恰恰是所有 XSUB 都在场的情形，在学完本节以后，你就能够完全理解由 SWIG 和 *xsubpp* 所产生的代码了。

首先让我们先来搞清楚 Perl 是如何发现你的 XSUB 的。也就是说，如果有人在脚本中书写有“`add($a, $b, $c)`”，那么 Perl 是怎么知道去调用 C 过程 `add`，或 `my_add` 或是别的什么例程的呢？其实，你必须像下面这样使用 `newXS`，创建一个子例程名（我们在脚本空间中这么来称呼它）与一个 C 过程之间的联编：

```

extern XS(add); /* 宏 XS，在下面的表 20-7 中有解释 */
newXS("add", add, "add.c"); /* 用于调试用的文件名 */

```

对于一个名为 `foo` 的模块，XS 和 SWIG 会产生一个称做 `boot_foo` 的过程，它使用 `newXS` 来将所有那个模块中的所有 XSUB 与对应的名字进行联编。这种方式的雅致之处在于 `boot_foo` 本身也是一个 XSUB，而且如果你使用动态加载的话，这个过程将由动态加载器模块在运行时刻进行调用。

XSUB 使用表 20-7 中所列的宏（在 *XSUB.h* 中定义）来检查堆栈和返回结果。

表 20-7 用于操纵堆栈的宏（解释器的嵌入）

函数 / 宏	描述
XS	<p>提供 XSUB 所需要的标准原型。例如，过程 <code>foo</code> 的声明应如下：</p> <pre> XS(foo) {     } </pre>



表 20-7 用于操纵堆栈的宏（解释器的嵌入）（续）

函数 / 宏	描述
DXSARGS	定义一些由其他宏使用的局部变量。其中重要的一个就是名为 <code>items</code> 的整数，它包含了由调用者推送到堆栈上的参数个数。
SV* ST( <i>n</i> )	从堆栈中获取第 <i>n</i> 个参数（它是一个 SV*）。ST(0) 表示第一个参数 ( <code>\$_[0]</code> )，而 ST( <code>items-1</code> ) 表示最后一个参数。
XSRETURN( <i>n</i> )	表明你已经在堆栈上留下 <i>n</i> 个结果参数然后返回。在通常你只有一个值要返回的情况下，你可以使用下面列举的更方便的宏。
XSRETURN_NO	在堆栈上留下一个值为 0, 1, 或 undef 的 SV 以后执行 XSRETURN(1)。
XSRETURN_YES	
XSRETURN_UNDEF	
XSRETURN_EMPTY	与 XSRETURN(0) 相同。
XSRETURN_IV ( <i>int</i> )	留下一个新的具有相应值类型的 mortal 标量变量。当调用者执行 FREETMPS 时这个标量变量将被删除。
XSRETURN_NV ( <i>double</i> )	
XSRETURN_PV ( <i>char *</i> )	

下面的代码段描述了手工编制的 XSUB `add`，它将所有的输入参数相加后返回结果：

```
#include <perl.h>
#include <XSUB.h>
XS(add)                                /* 所有的 XSUB 都有这种原型 */
{
    int sum = 0;
    dXSARGS;                           /* 定义 'items', 并将其初始化 *
                                        * 参数的个数 */
    if (items == 0)
        XSRETURN_IV(0);                /* 如果参数列表为空就返回 0 */
    for (--items ; items >= 0 ; --items) {
        if (SvIOK(ST(items))           /* 如果 SV 包含有一个整数 */
            sum += SvIV(ST(items));
    }
}
```

```

        XSRETURN_IV (sum);
    }

```

## 返回一组变长的结果

前面例子中的子例程返回了一个参数。返回多个参数也不复杂。下面的例子描述了将一个，它将空字符结尾的字符串数组 (argv) 在堆栈上转换成同等个数的结果参数：

```

int i = 0;
for ( ; *argv; argv++, i++) {
    ST(i) = sv_2mortal(newSVPV(*argv, 0));
}
XSRETURN(i);

```

如你所见，返回参数占据了 ST(0) 与 ST(n-1) 之间的一段参数堆栈。XSRETURN 通过调整标记堆栈，来使调用者获得返回的标量变量的数量。需要注意的重要一点是，前面的这段代码并没有修改处于同一段堆栈中输入参数；它更新堆栈以使其指向新的 SV（要知道这个堆栈就是包含 SV\* 的数组）。要想直接修改一个输入参数，你就应该这么来写：

```

sv_setpv(ST(i), "hello", 0); /* 类似于修改 $_[i] */

```

尽管诸如 read 这样的函数是这么做的，我还是建议你避免使用这种功能，而是创建新的 SV 作为替代。此外，为了使调用代码免于操心内存管理或引用记数的问题，你应该通过把这些值设置为 mortal，将它们交由 Perl 来负责。在作用域终止时它们将会被自动删除。

## 要确保堆栈足够大

宏 ST 直接指向堆栈中的对应位置。因为堆栈有可能无法进行足够的扩展来容纳宏中的变元，所以你不能任意的编写出如 ST(100) 这样的宏，它极有可能导致系统的崩溃。宏 EXTEND 将确保堆栈拥有足够大的空间来容纳你的数据：

```

EXTEND(sp, 100); /* 将堆栈扩充 100 个元素 */

```

这个宏在调用者与被调用的子例程中都能使用。变量 `sp` (堆栈指针) 是自动为你定义的 (由 `dSP` 和 `dXSARGS` 定义)。本应该由 `ST()` 通过使用 `av_store()` 来自动的扩展堆栈, 但是那样会大大的降低运行速度。

还有另一种解决方案。如果我们复位堆栈指针使其我们这段堆栈的底部, 那么我们就可以使用能够自动扩展堆栈的宏 `XPUSH` 了:

```
i = 0;
sp -= items;          /* 重置堆栈指针到起始位置 */
for ( ; *argv; argv++, i++) {
    /* 将新的拥有字符串值的 mortal 标量变量推送到堆栈上 */
    XPUSHs(sv_2mortal(newSVpv(*argv, 0)));
}
XSRETURN(i);
```

这恰恰是 XS 中的 `PPCODE` 指令所采用的策略, 这一点我们马上就要看到。我前面已经说过, 这种代码并不修改输入参数; 它只是简单的将堆栈中的那些指针替换成新的。要注意, 如果我们忘记复位栈指针的话, 我们就会将内容堆积在输入参数的上面, 那么在过程返回时就会搞得一团糟。

## 其他堆栈的内幕

让我们来简要的看一看 Perl 中可用堆栈的 (除了参数栈和标记栈) 情况, 这将有助于我们理解前面几节中所讲的宏在内部都做些什么工作。除非你对这些种类的细节感兴趣, 否则你完全可以跳过这一节而不会损失连贯性。

### 存储堆栈 (savestack)

这个堆栈被用做存储在套作用域中有可能改变的所有全局信息的仓库。例如, 为了安全的存放一个整数, Perl 使用了一个称做 `SSPUSHINT(scope.h)` 的宏。这个宏将三条信息推入 `savestack` 堆栈中: 整数的值, 整数的地址, 以及整数已经被存储的标识。此时该整数值就可以在嵌套域中进行任意的改变了。在当前作用域结束时, Perl 从存储堆栈中弹出所需数据, 并且还知道由于一个整数已经进行了存储, 它必定还存储了原先的指针和值。于是原先的整数就得以有效的恢复。

诸如 `local($a)` 这样的语句，就是通过将与“a”对应的 GV 及其标量变量值保存到存储堆栈内来实现的；标量变量值将被替换为一个新的标量变量。当作用域结束时，那个 GV 与其标量变量指针将会被自动恢复。

#### 作用域堆栈 (scopestack)

作用域堆栈用于记住不同作用域（这与 `markstack` 为变元堆栈提供书签标记类似）所对应存储堆栈中的位置。当作用域结束时（在执行了 `LEAVE`），Perl 会准确的知道需要从存储堆栈中弹出多少对象并将其恢复成原先的数值。

#### 临时变量堆栈 (tmps\_stack)

当你创建了一个 mortal 变量或是将一个变量标记为 mortal（使用 `sv_2mortal` 或在脚本空间中使用 `local`）时，Perl 就会将该 SV 推入这个堆栈中（而不会影响它的引用记数）。作用域结束时，它会递减所有在那个作用域中推入此堆栈的临时变量的引用记数。回想一下 `my` 变量（词法变量）是存储在特定于 CV 的便签簿中的，因此它们与临时变量堆栈没有任何关系。

#### 返回堆栈 (retstack)

在调用一个子例程前，Perl 通过将跟在那个子例程调用后的起始操作码推入 `retstack` 的方式来记住它。

#### 上下文堆栈 (cxstack)

这个堆栈跟踪保留当前代码块的上下文信息，如块标识和当 `last`、`redo` 或 `next` 被调用时要执行的 CV 等信息。在该代码块退出时它们将被恢复成原先的元素。我所不明白的是为什么要使用两种堆栈来处理与作用域有关的上下文信息。

## 内涵丰富的扩展

我们现在已经彻底的掌握了所需要的信息，并且还手工编制了几个扩展，现在我们做好了充分利用 SWIG 和 XS 的准备。在这一节中我们先来看看由 XS 所产生的代码类型。其实，SWIG 几乎产生相同的代码，因此这些解释对这两种工具都是充分的。然后我们将编写类型映射和小段代码来帮助 XS 处理 C 结构，使用 Perl 对象来包裹 C 结构以及最终与 C++ 对象进行连接。大多数的讨论也与 SWIG 有关，这也是我们只研究了一个 SWIG 例子的原因。那也就是说下面几页内容中描述的

特定 XS 类型映射的例子，可以让使用 SWIG 轻松的得到解决，而无须用户定义的类型映射。

## XS 扩展剖析

要想理解 XS 类型映射以及诸如 CODE 和 PPCODE 等关键词的作用，来仔细了解一下由 *xsubpp* 所产生的粘连代码还是值得的。考虑下面模块 Test 的 XS 声明，它包含了一个接受两个参数并返回一个整数的函数：

```
MODULE = Test  PACKAGE = Test
int
func_2_args(a, b)
    int    a
    char*  b
```

*xsubpp* 将它翻译成下面的代码（斜体形式的注释是后来加上的）：

```
XS(XS_Test_func_2_args) /* 使用包名来重置函数名 */
                        /* 增加来保证其唯一性 */
{
    dXSARGS;             /* 声明 "items", 并使用 */
    if (items != 2)       /* 堆栈上的条目数来进行初始化 */
        croak("Usage: Test::func_2_args(a, b)");

    { /* 开始一个新的代码块来允许变量声明 */
        /* 内建类型映射将堆栈翻译成 C 变量 */
        int    a = (int)SvIV(ST(0));
        char*  b = (char *)SvPV(ST(1),na);
        /* RETVAL 的类型与函数返回相匹配 */
        int    RETVAL;

        RETVAL = func_2_args(a, b);
        ST(0) = sv_newmortal();

        /* 输出的类型映射将 C 变量翻译到堆栈 */
        sv_setiv(ST(0), (IV)RETVAL);
    }
    XSRETURN(1); /* 使 Perl 知道有一个返回参数已经进入堆栈 */
}
```

这些与我们在“被调用方：手工编制XSUB”一节中所学的代码实际上是相同的。请注意堆栈上的参数是如何转换成两个参数 *a* 和 *b* 的。接着 XS 函数调用了实际的 C 函数 `func_2_args`，获取其返回值并将结果打包返回参数堆栈。

现在让我们添加一些常用的 XS 关键词来看看它们是如何由 *xsubpp* 所采用的。XS 代码片段如下：

```
int
func_with_keywords(a, b)
    int    a
    char*  b
    PREINIT:
        double c;
    INIT:
        c = a * 20.3;
    CODE:
        if (c > 50) {
            RETVAL = test(a,b,c);
        }
    OUTPUT:
        RETVAL
```

将被翻译成下面的这些代码：

```
XS(XS_Test_func_with_keywords)
{
    dXSARGS;
    if (items != 2)
        croak("Usage: Test::func_with_keywords(a, b)");
    {
        int    a = (int)  SvIV(ST(0));
        char*  b = (char *)SvPV(ST(1),na);
        double c;
        int    RETVAL;
        c = a * 20.3;
        if (c > 50) {
            RETVAL = test(a,b,c);
        }
        ST(0) = sv_newmortal();
        sv_setiv(ST(0), (IV)RETVAL);
    }
}
```

```

    }
    XSRETURN(1);
}

```

如你所见，在 PREINIT 中所提供的代码就跟在类型映射之后，以确保在主代码开始以前所有的声明都是完整的。所处的位置对于传统的 C 编译器很重要，但是对 C++ 编译器来说就不成问题，因为它允许在代码块中的任何位置进行变量声明。INIT 段则被插入在自动生成的对函数的调用之前，这里就是指在 CODE 段开始以前。CODE 指令为我们提供了插入任何代码段的灵活性。如果没有它，*xsubpp* 就得像我们在前一个例子中所见到的那样，简单的插入一个对函数 `func_with_keywords(a,b)` 的调用。

关键词 CODE 就像一个典型的 C 调用：你可以修改输入参数，并可以返回至多一个参数。要想处理不定数量的输入参数或输出结果的话，你就需要使用关键词 PPCODE。为了演示一下 PPCODE 的实现，请考虑一个函数 `permute`，它接受一个字符串，计算出它所有的排列并返回一个动态分配的字符串数组（以空字符结尾的 `char**`）。假定我们在 Perl 中要像下面这样来存取它：

```
@list = permute($str);
```

我们之所以要在这里使用 PPCODE 是因为该函数要求返回不定数量的标量变量。下面的小段代码描述了 XS 文件的内容：

```

void
permute(str)
    char *    str
PPCODE:
    int i = 0;

/* 调用 permute。它返回一个空字符结尾的字符串数组 */
    char ** ret = permute (str);

/* 将这些参数拷贝到 mortal 标量变量，并将其推入
 * 堆栈 */
    char **p = ret;
    for (; *p; p++, i++) {
        XPUSHs (sv_2mortal(newSVpv(*p, 0)));
    }
    free(p);

```

```
XSRETURN(i);
```

它将被翻译成下面的代码：

```
XS(XS_Test_permute)
{
    dXSARGS;
    if (items != 1)
        croak("Usage: Test::permute(str)");

    /* PPCODE 调整堆栈指针 (CODE 并不完成此类工作) */
    SP -= items;

    {
        char * str = (char *)SvPV(ST(0), na);
        int i = 0;
        /* 调用 permute。它返回一个空字符结尾的字符串数组 */

        char ** ret = permute (str);
        /* 将这些参数拷贝到 mortal 标量变量，并将其推入
         * 堆栈 */
        for ( ; *ret ; ret++, i++) {
            XPUSHs (sv_2mortal(newSVpv(*ret, 0)));
        }
        free(ret);
        XSRETURN(i);
        PUTBACK;          /* 这两条语句是冗余的 */
        return;           /* 因为 XSRETURN 这两项工作都完成了 */
    }
}
```

PPCODE 指令与 CODE 存在一个小的但却意义重大的不同之处：对于这个函数调用，它调整了堆栈指针 SP 而使其指向 Perl 堆栈帧的底部（也就是指向 ST(0)），从而可以使我们使用宏 XPUSH 来扩展堆栈来推送任意数量的参数（请回想一下我们在“确保堆栈足够大”一节中的讨论）。我们马上就会看到为什么我们使用类型映射就不行。

## XS 类型映射：介绍

一个类型映射就是一小段用于将参数栈上的标量变量值与对应的 C 标量变量实体



(整数, 双精度浮点数, 指针) 进行相互转换的代码。一个类型映射只能应用于一个方向。这里需要强调的重要一点是, 对于一个类型映射来说, 其输入和输出均为各自领域中的标量变量。比如, 你无法让一个类型映射接收一个标量变量, 却返回一个 C 结构。然而你却可以返回一个指向这个结构的指针。这也是前面一节中的 `permute` 例子为什么不能使用类型映射的原因。我们可以编写一个类型映射来把一个 `char**` 转换成指向数组的引用, 然后将其交给脚本编写人员来对它进行间接访问。在 SWIG 中由于不支持与 `PPCODE` 等价的机制, 所以这将是唯一的选择。

类型映射的另一个限制就是它们每次只能转换一个参数, 你不能基于多个输入参数来做出判定, 正如我们在第十八章“扩展 Perl: 第一课”中所提到的那样 (“如果参数 1 为 'foo', 那么把参数 2 加 10”)。XS 则提供了 `CODE` 和 `PPCODE` 指令来帮助你解决了这个问题, 而 SWIG 则不行。但是请回想一下第十八章中的“自由度”一节中所提到, 两个 SWIG 的限制可以在脚本空间中进行简单而有效的处理。

尽管 `xsubpp` 能够为普通类型的 C 参数提供转换操作, 但是对于所有用户定义类型我们还必须要编写定制的类型映射。假设我们有一个包含下面两个函数的 C 库:

```
Car*  new_car();
void  drive(Car *);
```

我们想在 Perl 中对它们这么来存取:

```
$car = Car::new_car;
Car::drive($car);
```

让我们首先来为该问题编写 XS 文件:

```
/* Car.XS */
#include <EXTERN.h>
#include <perl.h>
#include <XSUB.h>

#include <Car.h> /* 不用关心 Car* 会是什么样子 */

MODULE = Car  PACKAGE = Car
```

```

Car *
new_car ()

void
drive (car)
    Car *    car

```

如你所见，我们需要两个类型映射：一个用于将 `Car*` 转换成 `$car` 的输出类型映射，以及一个用于反方向转换的输入类型映射。我们先来编辑一个名为 *typemap*（注 11）的类型映射文件，它包含三段内容：TYPEMAP、INPUT 和 OUTPUT，如下所示：

```

TYPEMAP
Car *      CAR_OBJ

INPUT
CAR_OBJ
    $var = (Car *)SvIV($arg);

OUTPUT
CAR_OBJ
    sv_setiv($arg, (I32) $var);

```

TYPEMAP 段为你有可能很复杂的 C 类型 (`Car*`) 创建了易于使用的别名（这里就是指 `CAR_OBJ`）。类型映射文件中的 INPUT 和 OUTPUT 段现在就可以使用这个别名了，它们还包含了用于将相应类型的对象转换成一个 Perl 值或反方向操作的代码。当在一个问题中使用了类型映射时，标记 `$arg` 将会被参数栈上相应的标量变量取代，而 `$var` 则由对应的 C 变量名来取代。在这个例子中，输出类型映射将向标量变量的整数域中填充一个 `Car*`（请回想一下我们在“SV 与对象指针”一节中的讨论）。

TYPEMAP 段中别名的好处就在于，多个类型可以映射到相同的别名上。也就是说，一个 `Car*` 和 `Plane*` 可以同时以 `VEHICLE` 为别名，而由于 INPUT 和 OUTPUT 段只使用别名，所以这两种类型将共享相同的转换代码。Perl 的发行版中包含了一个提供所有基本类型映射的类型映射文件（请看 `lib/ExtUtils/typemap`），你可以

---

注 11： 我们之所以选择这种特别的名称，是因为 `h2xs` 生成的 `makefile` 能识别它，并将它提供给 `xsubpp`。而且还允许从不同的目录中拾取多个类型映射文件。

自由的使用定义在该文件中的别名。例如，你可以使用别名 `T_PTR`（而不是 `CAR_OBJ`）而在相应的 `INPUT` 和 `OUTPUT` 段中使用那个别名。也就是说我们的类型映射文件可以简单的这么来写：

```
TYPEMAP
Car *      T_PTR
```

实际上 `T_PTR` 的 `INPUT` 和 `OUTPUT` 段与上面 `CAR_OBJ` 的完全相同。

## 使用 XS 进行对象连接

假设要为脚本编写人员提供书写下面这种代码的能力，而不对 C 库做任何改变：

```
$car = Car::new_car(); # 与以前一样
$car->drive();
```

换句话说，我们的类型映射中的 `OUTPUT` 段需要将一个 `Car*` 转换成一个经过 `bless` 的标量变量引用，如我们在“SV 与对象指针”一节中所讨论的那样。`INPUT` 段包含了反方向的转换：

```
TYPEMAP
Car *      CAR_OBJ

OUTPUT
CAR_OBJ
    sv_setref_iv($arg, "Car", (I32) $var);

INPUT
CAR_OBJ
    $var = (Car *)SvIV((SV*)SvRV($arg));
```

`sv_setref_iv` 为一个整数提供了一个新分配的 SV 并将第一个参数转换成一个引用，使其指向这个新的标量变量，并将它 `bless` 到相应的模块中（请参照表 20-1）。在这个例子中，我们把指针转换成一个 `I32`，并使函数认为我们提供的是一个整数。

## 使 XS 类型映射更加通用

前面例子中的类型映射只限于具有 Car 类型的对象。我们可以使用 TYPEMAP 段的别名机制来使这种类型映射通用化，从而能容纳任意的对象指针。考虑下面的类型映射，改动的地方用黑体显示：

```

TYPEMAP
Car *      ANY_OBJECT

OUTPUT
ANY_OBJECT
    sv_setref_pv($arg, CLASS, (void*) $var);

INPUT
ANY_OBJECT
    $var = ($type) SvIV((SV*)SvRV($arg));
  
```

我们所做的就是使别名，类型转换，以及类名通用化。`$type` 为当前 C 对象的类型（TYPEMAP 段中别名的左半部分），因此对于这种情况来说就是 `Car*`。因为我们想使类名通用，所以我们采用了在第七章“面向对象编程”中所使用的策略——让脚本用户使用箭头记号：

```
$c = Car->new_car();
```

这个调用将以模块名作为第一个参数，它是在 XS 文件的 CLASS 参数中获取的：

```

Car *
new_car (CLASS)
    char *CLASS
  
```

剩下的唯一一件事就是，我们想让用户使用 `Car->new` 来代替 `Car->new_car`。C 没有多态机制，并不表示脚本用户也不能有。关键词 CODE 可以简单的完成这项工作：

```

Car *
new (CLASS)
    char *CLASS
    CODE:
  
```

```
    RETVAL = new_car();  
OUTPUT:  
    RETVAL
```

Drive 方法无须做任何改动。

使别名通用化以后,我们就可以将别名ANY\_OBJECT应用到任何其他的对象上,只要它们也遵循某些方法中声明及初始化一个CLASS变量的约定就行,这些方法返回TYPemap段中所声明的类型的指针。在前面的例子中,初始化将被自动执行,这是因为 Perl 将类名作为第一个参数来提供。

## C++ 对象与 XS 类型映射

假定你有一个名为 Car 的 C++ 类,它支持一个构造函数和一个称做 drive 的方法。你可以像下面这样在 XS 文件中定义相应的接口:

```
Car *  
Car::new ()  
.  
void  
Car::drive()
```

在转换完所有的参数之后(如果有的话),*xsubpp* 会将声明 new 转换成一个等价的构造函数调用:

```
XS(XS_Car_new)  
{  
    dXSARGS;  
    if (items != 1)  
        croak("Usage: Car::new(CLASS)");  
    {  
        char * CLASS = (char *)SvPV(ST(0),na);  
        Car * RETVAL;  
        RETVAL = new Car();  
        ST(0) = sv_newmortal();  
        sv_setref_pv(ST(0), CLASS, (void*) RETVAL);  
    }  
    XSRETURN(1);  
}
```

与前一个例子不同，*xsubpp* 将会自动提供 CLASS 变量。然而你还会需要这些类型映射来将 `Car*` 转换成等价的 Perl 对象引用。`drive` 接口定义将被转换成下面的代码：

```
XS(XS_Car_drive)
{
    dXSARGS;
    if (items != 1)
        croak("Usage: Car::drive(THIS)");
    {
        Car *    THIS;
        THIS = (Car *) SvIV((SV*)SvRV(ST(0)));
        THIS->drive();
    }
    XSRETURN_EMPTY;
}
```

*xsubpp* 会自动产生指向对象的 THIS 变量。CLASS 及 THIS 都可以用在同一个 CODE 段中。

Dean Roehrich 的《XS Cookbooks》中提供了几个 XS 类型映射的优秀例子，所以在你开始编写自己的这套东西以前一定要去看一看。

## 使用 XS 进行内存管理

我们到目前为止都没有费神考虑内存管理的问题。在前面的例子中，函数 `new` 分配一个对象，接着由类型映射代码将其填充到一个标量变量值中。当这个标量变量超出作用域或被赋给其他的内容时，如果它还没有经过 `bless`，那么 Perl 就会忽略这个指针——考虑到它已经相信标量变量中包含的就是一个整数值，所以这并不奇怪。这必定会产生内存泄漏 (memory leak)。但是如果标量变量经过 `bless`，Perl 就会在它被清除时调用其 `DESTROY` 例程。如果这个例程像下面这样是在 XS 中编写的，它就为我们提供了删除所分配内存的机会：

```
void
DESTROY(car)
    Car *car
CODE:
    delete_car(car); /* 释放那个对象 */
```

C++ 接口更是简单:

```
void  
Car::DESTROY()
```

这个例子中, *xsubpp* 自动调用 “delete THIS”, 这里如我们前面所见到的, THIS 表示对象。

## 推荐的内存分配与释放例程

Perl 库中提供了一组用于替换常规的动态内存管理例程 (列在表的左半部分) 的函数和宏:

不要使用	使用
malloc	New
free	Safefree
realloc	Renew
calloc	Newz
memcpy	Move
memmove	Copy
memzero	Zero

这些替代函数 (宏) 使用由 Perl 所提供的 malloc 版本 (默认情况下), 而且还可以选择收集内存利用统计。推荐使用这些例程而不是常规的内存管理例程。

## SWIG 的类型映射

SWIG 产生了几乎同 *xsubpp* 一样的代码。因此, 你可以预期其类型映射也与 XS 的非常类似 (如果不是完全相同的话)。考虑一下前面讨论的 *permute* 函数。我们需要将 *char\*\** 转换成一个列表, 但是由于类型映射允许它们的输入及输出参数可以为标量变量, 于是下面的类型映射将它转换成一个列表引用:

```
%typemap(perl5,out) char ** { // 所有函数都返回 char **  
                                // 获取这个类型映射  
    // $source 类型为 char **
```

```

// $target 类型为 RV (指向一个 AV)
AV *ret_av = newAV();
int i      = 0;
char **p   = $source;
/* 首先分配一个新的 AV, 要使用正确的尺寸 */
while (*p++)
    ; /* 递增, p 当 *p 为非空时 */
av_extend(ret_av, p - $source);

/* 对于字符串数组中的每个元素, 都创建一个新的
   * mortal scalar, 并将其装入上面的数组中 */
p = $source;
for (i = 0, p = $source; *p; p++, i++) {
    av_store(ret_av, i, sv_2mortal(newSVPV(*p, 0)));
    p++;
}
/* 最后, 创建一个指向该数组的引用; 这个类型映射的“目标” */
$target = sv_2mortal(newRV((SV*)ret_av));
}

```

SWIG的类型映射特定于某种语言, 因此perl5 参数out 表示函数的返回参数, 而且这个类型映射应用于所有具有char\*\* 返回值的函数。\$source和\$target是具有恰当数据类型的变量: 对于一个类型映射in, \$source是一个Perl类型, 而\$target为相应函数参数所预期的数据类型。请注意, 与XS的\$args和\$var不同, SWIG的\$source和\$target根据类型映射的方向不同会切换它们的含义。

如果你不想让这个类型映射应用于所有返回char\*\*的函数, 你可以像下面这样确切的指定, 将它应用到哪个参数或是哪个函数:

```

%typemap(perl5,out) char ** permute {
    ...
}

```

请查阅 SWIG 的文档来了解其他许多与类型映射相关的功能。

## 简单的嵌入式 API

我们已经学习了足够的知识来实现在第十九章中引入的方便的 API。它们是



perl\_call\_va、perl\_eval\_va，以及用于存取或修改标量变量值的函数集：get\_int、set\_int等等。我们在这一节只实现perl\_call\_va。因为perl\_eval\_va不需要任何输入参数（要被eval的字符串包含了所有的信息），所以它只是这个过程的一种简单形式。那些修改标量变量的API函数均为简单的建立在sv\_set\*，av\_store和hv\_store上的包裹函数，所以作为一项练习留给读者来完成（注12）。

回想一下，perl\_call\_va接受一个以NULL结尾的类型参数的列表。这个列表中既包含了输入参数也包含了输出参数。下面的实现通过XPUSH输入参数，以及将输出参数存储在一个Out\_Param结构的数组中，来处理整个列表。知道了调用者所需要的输出参数个数允许我们来指定G\_SCALAR、G\_ARRAY或G\_DISCARD。整个代码如例20-3所示。

例20-3: perl\_call\_va的实现

```
#define MAX_PARAMS 20
typedef struct {
    char type;
    void *pdata;
} Out_Param;          /* 用来记住 "Out" 段 */

int perl_call_va (char *subname, ...)
{
    char      *p    = NULL;
    char      *str = NULL; int i = 0; double d = 0;
    int        nret = 0;          /* 期望返回的参数个数 */
    int        ii   = 0;
    va_list    vl;
    int        out = 0;
    int        result = 0;
    Out_Param op[MAX_PARAMS];

    dSP;                                /* 标准的 ... */
    ENTER;                              /* ... 开端 */
    SAVETMPS;
    PUSHMARK(sp);
    va_start (vl, subname);
    while (p = va_arg(vl, char *)) { /* 获取下一个参数 */
```

注12： 我总想那么说！（请参看序言来了解可以下载该代码以及书中其他例子代码的FTP站点。）

```

switch (*p) {
case 's':                                /* 字符串 */
    if (out) {
        /* 如果我们处理 "Out" 段就会运行到这里 */
        op[nret].pdata = (void*) va_arg(vl, char *);
        op[nret++].type = 's';
    } else {
        str = va_arg(vl, char *);
        ii = strlen(str);
        XPUSHs(sv_2mortal(newSVpv(str, ii)));
    }
    break;
case 'i' :                               /* 整数 */
    if (out) {
        op[nret].pdata = (void*) va_arg(vl, int *);
        op[nret++].type = 'i';
    } else {
        ii = va_arg(vl, int);
        XPUSHs(sv_2mortal(newSViv(ii)));
    }
    break;
case 'd':                               /* 浮点数 */
    if (out) {
        op[nret].pdata = (void*) va_arg(vl, double *);
        op[nret++].type = 'd';
    } else {
        d = va_arg(vl, double);
        XPUSHs(sv_2mortal(newSVnv(d)));
    }
    break;
case 'o'
    out = 1;                             /* 输出参数开始 */
    break;
default:
    fprintf (stderr, "perl_eval_va: Unknown option '%c'\n",
             *p);
    fprintf (stderr, "Did you forget a trailing NULL ?\n", *p);
    return 0;
}
if (nret > MAX_PARAMS) {
    printf (stderr, "Can't accept more than %d return params\n",
            MAX_PARAMS);
    return -1;
}
}

```

```

    va_end(v1);
    PUTBACK;
    /* 所有输入参数推入堆栈, 而且 "nret" 中包含预计的 Perl 函数返回值的数量 */
    result = perl_call_pv(subname, (nret == 0)? G_DISCARD :
                                (nret == 1)? G_SCALAR :
                                G_ARRAY );

    /* 处理输出变量 */
    SPAGAIN;
    if (nret > result)
        nret = result;

    for (i = --nret; i >= 0; i--) {
        switch (op[i].type) {
            case 's':
                str = POPp;
                strcpy((char *)op[i].pdata, str);
                break;
            case 'i':
                *((int *) (op[i].pdata)) = POPi;
                break;
            case 'd':
                *((double *) (op[i].pdata)) = POPd;
                break;
        }
    }

    FREEMPS ;
    LEAVE ;
    return result;
}

```

## 未来展望

这一节中我将描述我们可以期待的几个令人感兴趣的東西, 它们可能会在下几个主要的 Perl 发行版中出现。

### 多线程解释器

Malcolm Beattie 已经发布了一个基于 POSIX 线程的早期版本的线程安全 Perl 解释器。(请在 Perl 5 Porters 文档中搜索 “thrperl”。) 这个修改的解释器并不是线程优化的; 也就是说它本身不使用线程 (与 Java 环境不同, 它对

于更新用户界面和垃圾收集分别使用了不同的线程)。它允许用户创建任意多的线程,并提供了诸如监控器和条件变量等标准线程原语。现已实现的是,本章介绍的所有的全局数据结构只是简单的成为一种针对于每个线程的实体。也就是说,每个线程都有它自己的一套堆栈、符号表散列和线程局部变量如`errgv($@)`。词法变量独立于子例程和特定线程的便签簿进行分配。

### 静态类型暗示

通过给解释器提供暗示来获得更好的优化和类型检查。Larry 在 Perl 5 列表中的这个例子,其地位公认为接近于“hello world”,如下所示:

```
my Dog $spot = new Dog;
```

现在 `$spot` 在编译时被标记属于一个称做 `Dog` 的类,因此一个 `$spot->meow()` 这样的调用将会产生编译错误,除非是你的狗不正常。

### 更快速的对象

预期会更好的支持对象,并且有可能提供替换 `ObjectTemplate` 模块的新的标准。你可以像下面这样来写:

```
package Dog;
use Fields qw(breed color);
$spot = new Dog;
print $spot->{color};
```

这真有点像一种散列表存取,实际上它有可能在编译时,将属性名替换成那个字段的偏移量,而优化成对数组的存取。也就是说, `$spot->{color}` 成了 `$spot->[1]`。

### Perl 编译器

Malcolm 还提交了一个 Perl 编译器扩展,在编写这本书时还处于初始阶段。可以要求它将一个脚本程序翻译成 C 代码,而它又可以被编译成一个可执行文件。实际上这个可执行文件的速度不及脚本的解释执行速度,这是因为大多数工作仍然像现在这样是在操作码函数中完成的。静态变量暗示可以引入一些相当棒的优化。例如,如果你这么写:

```
my integer $i;
```

那么编译器将可以使用 C 的本地整数类型,而不是一个 `SV` —— 这将会加速循环和算述表达式的执行。

与 Python 和 Java 类似，编译器还可以产生一个字节代码文件，然后让解释器进行 `eval`。它还能够支持比现在 `-D` 所提供的更好的调试选项。

## 相关资源

如果这一章还没能让你解渴的话，下面就是些获取进一步信息的去处：

1. `perlguts` documentation. Jeff Okamoto 等。

`perlguts` 文档与 `perlembed` 和 `perlcall` 一起包含了极为丰富的内容。这个文档中的 API 参考 包含了这一章中讲述的许多函数和宏。

2. Perl 5 Porters 新闻入口及文档。

`perl.porters-gw` Usenet 新闻组是通往 Perl 5 Porters 邮件清单的大门，其中有大量有关 Perl 内部工作的讨论内容，甚至超过了有关移植的问题。经过文档中的文章可以通过地址 <http://www.rosat.mpe-garching.mpg.de/mailling-lists/Perl5-Porters/> 来进行检索。

3. XS Cookbooks. Dean Roehrich。

这些指导教程可以从 CPAN (请在 `authors/Dean_Roehrich` 目录下查找) 获得，其中提供了对覆盖所有 XS 功能的样例问题的解决方案。

4. `sfio` (Safe/Fast I/O Library). David Korn and Kiem-Phong Vo。

更快速，可扩展，而且是整体上比 `stdio` 更好的另一种解决方案，向后兼容 `stdio`。可以从 CPAN 的 `Misc` 目录下获取，或查看下面的 URL 来了解概要内容：<http://www.research.att.com/sw/tools/reuse/packages/sfio.html>。

5. Perl 编译器。

从 CPAN 的 `authors/Malcolm_Beattie/` 目录下获取。

路向前绵延又绵延  
现在已经走了很远很远，  
只要能，我就必须走下去  
走着，带着期盼的脚步  
直到它并入宽阔的大道  
在那里，许多的小路与使命交汇  
该去往何方？我无法回答。

— J.R.R. Tolkien

《The Lord of the Rings》

鄧平知覺

PDG

# 附录一

## Tk 组件参考

任何系统中最缺乏灵活性的部件就是用户。

——Lowell Jay Arthur

这个附录讲述了最为常用的Tk组件的属性和方法。请参阅Tk发行版中丰富的联机文档来了解更详细的内容。

表 A-1 展示了几乎被所有组件所共享的属性和方法。

表 A-1 通用组件属性

属性	描述
font	见第十四章“字体”一节的讨论。
background, foreground	名称 (“red”) 或 RGB 值 (“#FF00FFA”)。选项还可以被简写为 bg 和 fg。见第十四章“颜色”一节的讨论。
text	在组件中显示的字符串。使用上面所指定的前景颜色和字体进行显示。
image, bitmap	指定要在组件中显示的位图。请参阅第十四章“图片”一节中的有关创建与管理位图的讨论。
relief	边界风格：为 raised, sunken, flat, ridge 或 groove 中的一个。同时应当指定一个非 0 的 borderwidth 选项。
height, width	除标签组件之外，高度和宽度通常以像素为单位。对于标签组件，则是以字符数为单位（那个组件字体的平均字符宽度与高度的倍数）。
textvariable	指定变量的名字。当一个组件的值发生改变时，该变量就会被更新，反之一样。

表 A-1 通用组件属性 (续)

属性	描述
anchor	指定组件或其内部的信息如何进行摆放。必须为 n, ne, e, se, s, sw, w, nw 或 center 中的一个。例如, 一个值为 “nw” 的 anchor 将会告诉标签组件将其标签文本显示在它的左上角处。
方法	
configure()	一次可以改变许多属性: \$widget->configure ('bg' => 'red', 'width' => 20);
cget()	获取给定选项的当前值: \$color = \$widget->cget('bg');

注意, 诸如 text 和 textvariable 的属性并不是适用于所有的组件; 例如, text 对于滚动条来说没有任何意义。您可以有选择的在属性前面加上连字符 (这对于在 Tcl/Tk 中是必需的, 而在 Perl/Tk 中则是可选的)。

## 按钮 (Button)

按钮就是带有一个附加属性 command 选项的标签, 见表 A-2。我们前面已经提到过, 这些属性是对表 A-1 中许多属性的补充。

表 A-2 按钮的方法和属性

属性	描述
command	指定一个指向 Perl 子过程的引用, 当鼠标按钮 1 (默认情况下) 在该按钮上被释放时就会调用这个子过程。
width, height	指定以字符为单位的宽度和高度。
方法	
flash()	简短的切换颜色来使按钮闪动。
invoke()	如果存在的话就调用与按钮关联的 Perl 子过程。



## 单选按钮 (Radiobutton)

表 A-3 单选按钮的属性和方法

属性	描述
command	指定一个指向 Perl 子过程的引用，当鼠标按键 1（默认情况下）在单选按钮上释放时，该子过程将被调用。在 command 被调用以前，与“variable”关联的变量将被更新。
variable	存有一个变量的引用，并当按钮被点击时使用“value”属性的值更新它。反之，当它被更新并与“value”属性具有相同的值时，就会选中这个按钮（否则，对于其他别的情况就使其处于非选中状态）。
value	指定每当该按钮被选中时要保存在按钮相关的变量中的值。
<b>方法</b>	
select()	选中该单选按钮并将关联变量设置为与此组件对应的值。
flash()	简短的切换颜色来使按钮闪动。
invoke()	调用与按钮相关的 Perl 子例程，如果有的话

## 复选按钮 (Checkbutton)

表 A-4 复选按钮的属性和方法

属性	描述
command	指定一个指向 Perl 子过程的引用，当鼠标按键 1（默认情况下）在按钮上释放时，该子过程将被调用。在 command 被调用以前，与“variable”关联的变量将被更新。
variable	存有一个变量的引用，并根据指示器的状态使用“onvalue”或“offvalue”属性的值更新它。反之，当其被更新时，它就将自己同这些值进行匹配，并相应的使自己处于选中或非选中状态。
onvalue, offvalue	根据这些值与 variable 值的匹配情况来切换指示器的状态。它们分别默认为 1 和 0。

表 A-4 复选按钮的属性和方法 (续)

属性	描述
indicatoron	如果为假, 它就不会显示指示器。代之以切换整个组件的“relief”属性 (这会使它看起来就像一个按下去的按钮)。
<b>方法</b>	
select()	选中复选按钮并将关联变量设置为“onvalue”的值。
flash()	简短的切换颜色来使按钮闪动。
invoke()	如果有的话就调用与 command 属性关联的 Perl 子过程。
toggle()	切换按钮的选中状态以及 variable 的值。

## 画板 (Canvas)

表 A-5 描述了画板组件所支持的、以及使用组件的 `itemconfigure()` 方法进行配置的每个元素的属性。该组件本身并没有什么特别令人感兴趣的属性。然而在表的后半部分描述的方法则是组件的, 即便是其中的大多数都应用于单一的画板元素类型。请注意, 所有以一个元素 ID 为参数的方法, 都能够以一个预先配置的标签名为参数。

表 A-5 Canvas 类

元素属性	描述
<b>共有属性</b>	
fill, outline	用于填充区域和轮廓的颜色
tags	一组字符串的列表。这些是应用于该元素的标签。您可以使用 <code>addtag</code> 来向列表中增加更多的项。
stipple, outlinestipple	使用点刻法来绘制内部或轮廓。bitmap 的值用于指定点刻图案。
width	轮廓线的宽度。
<b>Arc</b>	
start, extent	逆时针方向移动的角度。

表 A-5 Canvas 类 (续)

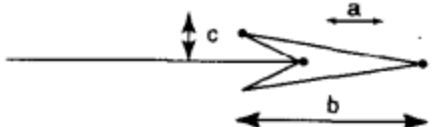
元素属性	描述
style	pieslice, chord, arc。对于最后一种情况, fill 选项将被忽略。
<b>Bitmap</b>	
anchor	见前面描述的组件属性。
bitmap	显示的位图。
background, foreground	每个位图像素的颜色。
<b>Image</b>	
anchor	见前面描述的组件属性。
image	显示的图片。
<b>Line</b>	
arrow	first, last, both 或 none。要绘制箭头的末端。
arrowshape	一个指向包含有像下面这样三维 a,b,c 列表的引用:
	
fill	颜色。
smooth, splinesteps	如果为 1, 就会绘制一条贝塞尔曲线来代替多边形。每个样条都使用 splinesteps 的数目进行逼近。
<b>Polygon</b>	
Smooth, splinesteps	见上面的“Line”。
<b>Oval</b>	
标准条目属性	
<b>Rectangle</b>	
标准条目属性	

表 A-5 Canvas 类 (续)

元素属性	描述
<b>Text</b>	
text, anchor	根据 anchor 来摆放 text。
justify	文本对齐: left, right 或 center。
<b>Window</b>	
Window	指定与该条目关联的组件。这个组件必须已经以该画板子元素的身份被创建。
<b>方法</b>	
create (type, x, y [x1, y1],[options...])	type 可以为上面所提到的条目类型中的一种 (不要大写)。返回一个唯一的整数 ID。
itemconfigure (ID, options..)	对上面所提到的一个或多个参数进行配置。
addtag, dtag	为条目增加标签以及删除标签。请参阅 Tk 文档中的标签规范。
bind	见第十四章中“事件联编”一节的讨论。
coords (ID [x0, y0 ...] ), move (id, xamount, yamount)	将条目移动到新的位置。coords 是一种绝对地址形式的移动, 而 move 则是相对于当前的位置进行移动。
delete (ID, [ID, ...] )	删除与标签或 ID 相对应的条目 (或一组条目)。
find (searchCommand? arg ...? )	检索所有符合某一约束的条目。约束的形式为: “above \$id,” “all,” “below \$id,” “closest x y,” “enclosed x1 y1 x2 y2,” “withtag id,” 等等。
postscript (?option value option value ...? )	为画板的一部分或整体产生 PostScript 的表达形式。请参阅 Tk 的画板文档来获取有关 postscript 的绘制选项。
raise, lower	提升或降低该条目。
scale (ID, xOrigin, yOrigin, xScale, yScale )	重新缩放由 ID 指定的条目。

## 文本框组件 (Text)

文本框组件方法中的许多都需要接收一个或多个索引为参数。索引可以是绝对数 (“base”) 也可以是相对数 (“base” + 修正量)。这两种类型的索引均以字符串的形式来指定。一些常用的基索引如下:

*line.char*

表示位于 *line* 行的第 *char* 个字符。为了与其他的 Unix 程序保持一致, 行编号使用了从 1 开始的这种模式。每一行中, 字符从 0 开始编号。

*end*

表示文本的末尾 (紧接最后换行符的字符位置)。

*insert*

插入光标当前所在的位置。

*mark*

表示正好处于名为 *mark* 的标记之后的字符位置。

*tag.first, tag.last*

表示一个标签的第一个和最后一个字符位置。

这些绝对的定位可以使用下面的一个或多个限定语来进行修正:

*+count chars, -count chars, +count lines, -count lines*

通过 *count* 数量的字符或行来调整基索引。

*wordstart, wordend, linestart, lineend*

调整索引以使其指向单词的第一个字符, 由索引指定的行起始位置或是紧跟单词或行的末尾位置。

表 A-6 描述了一些更为有趣的文本框组件的属性和方法。

表 A-6 文本框组件的属性和方法

文本	描述
tabs	为窗口指定一组制表标记，它是一个指向字符串列表的引用。每个字符串为一个跟有“l”，“c”或“r”（左，中，右——用以指定文本相对制表栏对齐的方式）的数字。
height,width	以字符为单位指定高度和宽度。
state	normal 或 disable。
方法	
文本操纵	
insert (index, {string, [tag]}+, )	在index处插入一个或多个字符串，还可以带有一个tag选项。index可以为前面所讲述的任何一种索引形式。
delete(index1, [index2])	删除位于index1的字符或是从index1到index2的区间。
get (index1, [index2])	获取位于index1的字符或是从index1到index2的区间。
see (index)	滚动组件以使index的位置可见。
search([switches], pattern, index, [stopIndex])	检索文本并返回与模式匹配的索引。在指定的情况下，检索终止于stopIndex。否则它就会回绕过来。switches包括forward, backward, exact（精确匹配——为默认情况），regexp, -nocase（忽略大小写），-count var（var为指向变量的引用，search将匹配的长度信息存储于其中）。switch列表由“--”终止。
索引	
index(index)	给定任何其他类型的索引返回一个line.col形式的绝对索引。
see (index)	确保处于index位置的文本可见。
markSet (markName, index)	为index指定一个逻辑书签。
markUnset (markName)	删除这个书签。

表 A-6 文本框组件的属性和方法（续）

文本	描述
<b>标签操纵</b>	
tagAdd (tagName, {index1. [index2]})+	为字符位置或区间指定增加标签。insert 是另一种标记文本的方式。
tagRemove (tagName, {index1. [index2]})+	将标签从指定的区域内去掉但是并不删除标签本身。
tagDelete	去掉并删除标签。
tagConfigure	配置标签的一个或多个属性。标签的属性如下所示。
<b>标签属性</b>	
-foreground, -background, -fgstipple, -bgstipple, -font	这些是常见的属性类型。尽量不要过多使用这些标签，不然文本看起来就会像赎金交纳条了——从不同的报纸或杂志上剪接下来的文字堆砌而成。
-justify,	center, left, right。
-relief, -borderwidth	应当同时指定边界宽度和背景以使浮雕效果显现出来。
-tabs	只有在那一行的第一个字符也属于同样的标签时才适用。
-underline	布尔选项。

## 输入条组件 (Entry)

输入条组件为简单的单行文本框组件。它们不支持标签、标记或嵌入式窗口。索引的语法因此也就简单一些：

*number*

其内容的索引，从 0 开始。

end

文本结束位置。

`insert`

紧跟在插入光标后面的位置

`sel.first, sel.last`

表示一个选择范围的第一个和最后一个字符。

表 A-7 中所有方法的 `index` 参数都接受上面所提到的任何一种样式的索引。

表 A-7 输入条组件的属性和方法

属性	描述
Show	如果为假，它会显示“*”，而不是真实的内容，常被用来获取口令信息。要注意，如果文本被选中后粘贴到其他的地方，真实的内容便会显现出来。
<b>方法</b>	
<code>get (index)</code>	获取整个字符串。
<code>insert (index, string)</code>	在 <code>index</code> 位置插入一个字符串。
<code>index (index)</code>	返回数字形式的索引。
<code>selectionFrom (index)</code>	用于设置或调整选择。
<code>selectionTo (index)</code>	
<code>selection (from,to)</code>	
<code>selectionClear</code>	
<code>selectionPresent</code>	如果存在一个选择就为真。

## 列表框组件 (Listbox)

在查看表 A-8 以前，让我们先来了解一下索引的语法，表中列举了列表框组件的属性和方法：

`number`

行索引，从 0 开始。

`end`

表示行的末尾。



`active`

定位光标当前所在的位置。活动位置处有下划线。

`anchor`

选择的定位点。

表 A-8 列表框组件的属性和方法

属性	描述
<code>height, width</code>	以行为单位的高度和以字符为单位的宽度。如果为0, 那么组件就会调整尺寸以容纳所有的元素。
<code>selectMode</code>	<code>single</code> , <code>browse</code> , <code>multiple</code> 或 <code>extended</code> 中的一个。
<b>方法</b>	
<code>get (index)</code>	获取整个字符串。
<code>insert (index, string)</code>	在 <code>index</code> 位置插入一个字符串。
<code>delete (index, [last])</code>	删除位于索引或区间中的内容。
<code>index (index)</code>	返回数字形式的索引。
<code>see (index)</code>	使第 <code>index</code> 个元素可见。
<code>selectionFrom (index)</code>	设置或调整选择。
<code>selectionTo (index)</code>	
<code>selection (from,to)</code>	
<code>selectionClear ()</code>	
<code>selectionPresent ()</code>	如果存在一个选择就为真。
<code>curselection()</code>	被选中的元素索引列表。

## 菜单 (Menu)

菜单按钮支持表 A-9 中所描述的方法。它们还支持表 A-10 中的方法, 并将那些调用指派给底层的菜单组件来执行。

表 A-9 菜单按钮组件的属性和方法

属性	描述
indicatorOn	如果为真，就在菜单项的右边显示一个菱形符号。
state	normal, active 或 disabled。
<b>方法</b>	
command	这些方法只是简单的指派给底层的菜单执行。请参阅表 A-10 中有关菜单对象的描述。
separator	
radiobutton	
checkboxbutton	
cascade	
menu	返回与该菜单按钮相关的底层菜单。

菜单组件的每个方法都理解下面的索引语法：

#### *number*

菜单项的索引，从 0 开始。当 tear-off 选项处于激活状态时，第 0 个菜单项将会成为由菜单自动插入的分割符。

#### *end, last*

表示最后的菜单项。

#### *active*

定位光标当前所在的位置。活动位置有下划线。

#### *none*

表示没有任何处于活动状态的菜单项。与 `activate()` 一起使用来激活所有的菜单项。

#### *pattern*

一个用于与所有菜单项进行匹配的模式。当前只保证精确匹配能够工作。

表 A-10 描述了菜单组件的属性以及每一种菜单项类型的可用选项。

表 A-10 菜单组件的属性和方法

菜单组件的属性	描述
<code>indicatorOn</code>	如果为真，就在菜单项的右边显示一个菱形符号。
<code>selectColor</code>	如果有指示器显示时，指示器的颜色。
<code>tearOff</code>	当它为真时，菜单的第 0 个菜单项为分割符。当您点击它时，菜单就会“撕下来”而成为一个分离的顶层窗口。
<b>菜单项属性外观</b>	
<code>foreground, background, font, image, indicatorOn, label</code>	常见的属性。注意这里是 <code>label</code> 而不是 <code>text</code> 。
<code>underline</code>	一个要加下划线字符的整数索引。
<code>accelerator keysequence</code>	表示一个要在标签右侧显示的字符串。与 Motif 不同，您必须自己来设置按键联编（通常在顶层窗口）。该选项只用于进行显示。
<b>执行动作与值</b>	
<code>state</code>	<code>normal</code> , <code>active</code> 或 <code>disabled</code> 。
<code>command</code>	适用于按钮类的菜单项，为要调用的子过程引用。
<code>value var</code>	适用于 <code>radiobutton</code> 菜单项。请参见表 A-3。
<code>variable var</code>	只适用于 <code>checkboxbutton</code> 和 <code>radiobutton</code> 菜单项。
<code>onval val, offval val</code>	适用于 <code>checkboxbutton</code> 菜单项。保存在关联变量中。
<b>方法</b>	
<code>command(options)</code>	创建相应类型的菜单项。每个菜单项都有自己的配置选项，如上所述。
<code>separator (options)</code>	
<code>radiobutton (options)</code>	
<code>checkboxbutton (options)</code>	
<code>cascade (options)</code>	
<code>add (type,options)</code>	上面的命令被转换成该调用。
<code>delete (index1, [index2])</code>	删除一个或一个区间的菜单项。

表 A-10 菜单组件的属性和方法（续）

菜单组件的属性	描述
insert (index1, type, options)	与 add 类似，但是是在所要求的索引位置进行插入。
entryconfigure (index, options), entrycget (index)	配置及获取菜单项的属性。

## 滚动条组件（Scrollbar）与滚动

滚动条支持表 A-11 中所列举的方法和属性。

表 A-11 滚动条的属性和方法

属性	描述
command	回调通常被用来改变关联组件的视图（也就是调用那个组件的 xview 或 yview）。
方法	
set(first,last)	它通常由关联的组件来调用用来告诉它有关组件当前视图的情况。 <i>first</i> 与 <i>last</i> 均为介于 0 和 1 之间的分数。值 0.2 和 0.6 将告诉滚动条组件正在分别显示文档的 20% 到 60% 之间的区域。
get	以列表（first, last）的形式返回滚动条的设置信息。

所有可以进行滚动的组件类型，除它们自身的（以及表 A-1 中所列举的通用内容）方法以外，还支持表 A-12 中的方法。

表 A-12 可滚动组件的属性和方法

属性	描述
xscrollincrement, yscrollincrement	如果指定了，滚动就可以以这些递增量为单位来进行。

表 A-12 可滚动组件的属性和方法 (续)

属性	描述
xscrollcommand, yscrollcommand	告诉组件当其在内部重新定位时要做的工作。通常该选项看起来会是下面的样子: \$scrollbar->configure
方法	
yview ('moveto' , <i>fraction</i> )	xview, yview以两种形式出现。如果第一个参数为“moveto”，那么组件将被指示以 <i>fraction</i> 代表最顶部 (或是对于 xview 时为最左边) 的行或像素的形式来改变它的视图。
yview('scroll', number, what) (xview 与之类似)	它指示组件根据增量的 <i>number</i> 来调整窗口中的视图是向上还是向下。what 指定这些增量是以 units 还是以 pages 的方式。如果 what 为 units, 那么它就是指上面所描述的增量属性。

## 标尺 (Scale)

表 A-13 标尺的方法和属性

属性	描述
command	指定一个指向 Perl 子过程的引用, 当标尺的值被改变时就会调用它。
variable	指向一个每当滑块移动时需要更新的变量。反之, 如果您想让滑块移动, 您就可以改变变量的值。
width, length	以像素为单位指定宽度和长度。注意, 它并不支持称做高度的属性。
orient	水平的还是垂直的。
from, to	实际数字。
resolution	如果大于 0, 显示和返回的值总是为该值均匀的倍数。默认为 1。
tickinterval	数字刻度之间的间距。如果为 0, 则不会显示任何刻度。
label, font	一个在滑块上部或左侧显示的标签 (依赖于标尺的摆放方向)。

表 A-13 标尺的方法和属性 (续)

属性	描述
方法	
set(value)	等同于将变量更新为给定的值。

## 树形列表组件 (HList)

这个组件是 Tix 发行版的一部分, Perl/Tk 像对待其他 Tk 组件一样也支持它。表 A-14 描述了它的属性。

表 A-14 Hlist 的方法和属性

属性	描述
command	指定一个指向 Perl 子过程的引用, 当在一个菜单项上双击时被调用。
drawbranch	如果为真, 就会绘制连接双亲与孩子及兄弟菜单项的线段。
browsecmd	每当一个菜单项上发生任何鼠标点击或拖曳事件时就会调用。
columns	列中的每一层缩进。列的尺寸可以被单独定制。
separator	分隔符字符。默认为 “.”。
selectmode	single, browse, multiple 或 extended。
indent	相继每一级缩进的数量, 以像素为单位。
方法	
add (entrypath, option,	创建一个新的菜单项。option 可以为处于 <i>position</i> , 位 <i>values</i> ) 于之前 <i>before</i> , 在之后 <i>path</i> , 所有的菜单项属性讨论如下。
delete (option, entrypath)	选项可以为 all, entry, offsprings, siblings。
column (col, width)	以像素或字符宽度的语意为单位设置宽度。
column (col, 'char', nchars)	

表 A-14 Hlist 的方法和属性 (续)

属性	描述
entryconfigure, entrycget	设置 / 获取下面讨论的菜单项属性。
info (option, entryPath)	选项可以为 children, exists, hidden, next, prev, selection, 等等。
hide, show	隐藏或者显示一个菜单项。
<b>条目属性</b>	
'itemtype'	text, imagetext 和 widget。
'text'	条目的标签。
'image'	一个位图或像素图。在 itemtype 为 imagetext 时使用。

注意，这里是指可滚动的组件，因此它还支持在表 A-12 中所列举的通用 xscrollcommand 与 yscrollcommand 属性。也就是说创建一个可滚动 Hlist 框的一种简单方法就是调用 `$parent->ScrlHList(options)`。



## 附录二

# 语法概要

因此 Ninjei 大师说：

“对于聪明人，只需一句话；

对于一匹千里马，只需轻轻的一鞭；

对于编写良好的程序，只需一条命令。”

——《The Zen of Programming (编程禅学)》

这个附录提供了对本书中使用的所有语法的精练描述。

## 引用

### 1. 简单变量的引用：

```
$ra = \ $a;      # 指向简单变量的引用
$$ra = 2;        # 简单变量引用的间接访问
$ra = \1.6;      # 指向常量的引用
```

### 2. 数组的引用：

```
$rl = \@l;        # reference to existing
$rl = [1,2,3];    # reference to anon. scalar
push (@$rl, "a"); # Dereference
print $rl->[3]     # 4th element of array pointed to by $rl
```

### 3. 散列表的引用：

```
$rh = \%h;        # 指向散列表的引用
$rh = {"laurel" => "hardy", "romeo" => "juliet"}; # 指向匿名散列表的引用
print keys (%$rh); # 间接访问
$x = $rh->{"laurel"}; # 用来抽取单个元素的箭头记号
@slice = @$rh{"laurel", "romeo"}; # 散列表片段
```



## 4. 代码的引用:

```
$rs = \&foo;           # 指向现有子过程 foo的引用
$rs = sub {print "foo"}; # 指向匿名子过程的引用
                        # (要记住最后的分号)
&$rs();                # 间接访问: 调用子过程
```

## 5. 普遍引用的间接访问操作。任何处于代码块中并返回一个引用的代码都可以进行间接访问:

```
@a = @{foo()};         # 对数组引用的间接访问
                        # 由 foo() 返回
```

## 6. 引用中的常见错误。下面所描述的所有代码都是错误的。在开发和测试时一定要使用 -w 命令行参数。

```
@foo = [1,3,4];        # 将数组引用赋值给一个数组
                        # 代之以使用圆括号

%foo = {"foo" => "bar"}; # 将散列表引用赋值给一个散列表
                        # 代之以使用圆括号

$foo = \($a, @b);       # 等同于 $foo = (\$a, \@b)
                        # 将一个枚举列表赋值给一个简单变量
                        # 将会返回最后一个元素 (因此,
                        # $foo 的内容为 \@b)。如果你需要的是一个
                        # 数组的引用, 就要使用 [ ]
```

## 嵌套数据结构

每个数组或散列表都是简单变量的集合, 这些简单变量中的一些或全部还可以是指向其他结构的引用。

## 7. 列表不是像下面这样进行嵌套:

```
@foo = (1, 3, ("hello", 5), 5.66);
```

要进行嵌套, 就得使第三个元素成为一个指向匿名数组的引用:

```
@foo = (1, 3, ["hello", 5], 5.66);
```

## 8. 一个嵌套数据结构的例子 (包含散列表的数组的散列表):

```
$person = { # Anon. hash
  "name" => "John", # '=>' 是 “,” 的另一种形式
```

```

"age" => 23,
"children" => [ # 匿名子数组
    {
        "name" => "Mike", "age" => 3,
    },
    {
        "name" => "Patty", "age" => 4
    }
]
];
print $person->{age}           ; # 打印出 John 的年龄
print $person->{children}->[0]->{age}; # 打印出 Mike 的年龄
print $person->{children}[0]{age} ; # 打印出 Mike 的年龄, 省略
                                # 下标之间的箭头

```

#### 9. 将上面的 \$person 进行格式化输出:

```

use Data::Dumper;
Data::Dumper->Dumper($person);
# Or,
require dumpVar.pl;
main::dumpValue($person);

```

## 闭包 (Closure)

#### 10. 闭包就是一种匿名子过程，它从包含它的环境中抓取词法变量。要记住它可不仅仅是获取匿名子过程刚一出现时的快照内容。

```

# 声明一个匿名子过程，并返回指向它的引用
my $foo = 10;
$rs = sub {
    print "Foo is $foo\n"; # Grabs $foo
};
&$rs(); # 通过引用调用这个闭包

```

#### 11. 即便是变量超出了作用域，闭包也会保留被抓取的变量值。

```

sub init_counter {
    my $num = shift; # 要抓取的词法变量
    $rs = sub { print $num++, " "; };
    return $rs;
}
$rs_counter = init_counter(10); # $rs_counter 为一个指向子过程的引用
for (1..5) {&$rs_counter()}; # 打印出 10 到 14

```

## 模块

12. 关键词package将开始一个新的名字空间（这会持续到另一条package声明的出现或是代码块结束）。所有用户定义的全局标识符（变量，子过程，文件句柄）都属于这个包。词法变量不属于任何包。

```
package Employee; # 放在文件 Employee.pm 中
@employees = ("John", "Fred", "Mary", "Sue");
sub list_employee { print @employees; }
1;                # 文件中最后一条执行的语句必须为
                  # 非零值，以表示成功加载
```

13. 加载模块 Employee:

```
use Employee;
#or
require Employee;
```

通过 -I 命令行选项，PERL5LIB 环境变量，@INC 来指定加载路径。

14. 使用全限定名来存取外部包的变量和子过程:

```
print @Employee::employees;
Employee::print();
```

没有强制的私有性。

15. 如果在那个包中没有发现某个子过程，默认子过程 AUTOLOAD 在存在的情况下就会被调用。\$AUTOLOAD 被设置为不存在子过程的全限定名。

16. 要使模块 C 继承模块 A 和模块 B，就需要使用超类模块的名称来初始化 C 的 @ISA 数组:

```
package A;
sub foo{ print "A::foo called \n";}
package C;
@ISA = ("A", "B");
C->foo();                # 调用 A::foo, 因为 B 不存在
```



## 对象

要点:

- 一个类就是一个包。Perl 没有用以定义对象模式的诸如 struct 或 class 的关键词。
- 你选择对象的表示形式 —— 对象模式不是由你来规定的。
- 构造函数没有特殊的语法。由你来选择用以分配对象并返回经过 blessed 或 typed 的指向对象的引用的子过程的名字。

17. 创建一个面向对象的包 —— 方法 1 (见下面的第 19 条)。

C++ 类为:

```
class Employee {
    String _name; int _age; double _salary;
    create (String n, int age) : _name(n), _age(age), _salary(0) {}
    ~Employee {printf ("Ahh ... %s is dying\n", _name)}
    set_salary (double new_salary) { this->_salary = new_salary}
};
```

成为:

```
package Employee;
sub create {
    # 分配及初始化
    my ($pkg, $name, $age) = @_;
    # 分配一个匿名散列表, 对其进行 bless, 并返回它.
    return (bless {name => $name, age=> $age, salary=>0}, $pkg);
}
sub DESTROY {
    # 析构函数 (类似于 Java 的 finalize)
    my $obj = shift;
    print "Ahh ... ", $obj->{name}, " is dying\n";
}
sub set_salary {
    my ($obj, $new_salary) = @_;
    $obj->{salary} = $new_salary; # 记住: $obj 为一个散列表引用
    return $new_salary;
}
```

## 18. 使用对象包:

```
use Employee;
$emp = Employee->new("Ada", 35);
$emp->set_salary(1000);
```

## 19. 创建面向对象的包 —— 方法 2 (见前面的第 17 条)。继承 ObjectTemplate, 使用属性方法来声明属性名, 并会自动得到构造函数 new 和属性存取函数:

```
package Employee;
use ObjectTemplate;
@ISA = ("ObjectTemplate");
attributes("name", "age", "salary");
sub DESTROY {
    my $obj = shift;
    print "Ahh ... ", $obj->name(), " is dying\n";
}
sub set_salary {
    my ($obj, $new_salary) = @_;
    $obj->salary($new_salary);
}
```

## 20. 类方法:

```
Employee->print();    # 1. “箭头记法”用于对象方法
new Employee ();      # 2. 类方法使用“间接记法”
```

这些类方法必须以包名作为第一个参数, 其后跟有剩余的参数。

## 21. 实例方法。下面是两种调用对象方法的方式:

```
$emp->promote();
promote $obj;
```

## 动态特性

## 22. 符号引用:

```
$i = "foo";
$$i = 10;           # 将 $foo 设置为 10
${"i"} = 10;        # 将 $foo 设置为 10
&$i();              # 调用 foo();
push (@$i, 10, 20); # 将 10, 20 推送到 @foo 中
```



## 23. 运行时表达式计算:

```
while (defined($str = <STDIN>)) {  
    eval ($str);  
    print "Error: $@" if $@;  
}
```

这是一个小巧的 Perl 外壳程序, 它从标准输入读取信息, 将 \$str 当作一个小程序, 并将编译和运行时错误放置在 \$@ 中。

## 24. 动态替换。对 s/// 操作符使用 /e 标志来指定一个表达式而不是匹配模式:

```
$1 = "You owe me 400+100 dollars";  
$1 =~ s/(\d+)\+(\d+)/$1 + $2/e;  
print $1; # 打印出 "You own me 500 dollars"
```

## 25. 模块与对象的方法调用 (见第 20 和 21 条):

```
$module->foo(); # 调用由 $module 所指示模块中的 foo()
```

## 例外处理

## 26. die 抛出一个例外, 而 eval 将其捕获。错误字符串被放置在 \$@ 中。下面的代码可能出现两种运行时错误:

```
eval {  
    $c = $a / $b; #1  
    die "Denominator cannot be negative" if ($b < 0); #2  
};  
print "Run-time error: $@";
```

在(1)位置 \$@ 的内容为 "Illegal division by zero", 在位置(2)为 "Denominator cannot be negative"。

## 元信息

## 27. 调用堆栈信息。使用 caller() 来查明是谁在调用这个子过程:

```
($package, $file, $line) = caller();
```

28. 包中全局变量的列表。对包 `Foo` 来说, `%Foo::` 包含了符号表, 其中的键值为全局标识符名, 而它们的值为 `typeglob`。
29. 查明一个引用所包含的内容。`ref($r)` 当 `$r` 为普通的简单变量时返回 `undef`, 当为一个指向简单变量的引用时返回 “SCALAR” (类似的有 “ARRAY”, “HASH”, “CODE” 和 “REF”), 当为一个经过 `bless` 的对象引用时返回包的名字。
30. 对象信息:

```
$obj->isa("FooV"); # 如果 $obj 继承自 Foo 就返回真值
$obj->can("bar"); # 如果它支持方法 "bar" 就返回真值
```

## Typeglob

31. `Typeglob` 赋值可以使标识符具有别名。在下面的例子中所有名为 `a` (简单变量, 数组, 散列表, 文件句柄, 格式) 的标识符也可以使用 `b` 来存取:

```
*a = *b ;      # 别名
$b = 10;       # 等价于修改 $a
b();           # 等价于调用 a()
```

32. 选择性的别名:

```
*a = \%b ;     # 只有 $a 成为 $b 的别名
```

33. 常量:

```
*a = \10;      # 将一个指向常量的引用的别名设置为一个 typeglob
$a = 20;       # 运行时错误 —— “企图修改只读变量”
```

## 文件句柄与格式

不存在直接对文件句柄或格式进行赋值的方式, 如作为参数传递给子过程, 将其保存在数据结构中或将其局部化等。作为替代, 我们使用相应的 `typeglob`。





# 词汇表

abstract class

抽象类

adaptor

适配器

API (Application Programming Interface)

应用程序编程接口

array

数组

assignment operator

赋值运算符

AST (Abstract Syntax Tree)

抽象语法树

atomicity

原子性

attribute

属性

AWT (Abstract Windowing Toolkit)

(Java 的) 抽象窗口工具包

base class

基类

binary large object(BLOB)

二进制大对象数据类型

bind

联编

bitmask

位屏蔽

block

阻塞

byte code

字节码

callback

回调函数

CASE (Computer Aided Software Engineering)

计算机辅助软件工程 (技术)

cast

类型转换

CGI (Common Gateway Interface)

公共网关接口

children

子 (节点, 指针)

class

类

class member

类成员

CLI (Call Level Interface)	调用级接口	durability	持久性
closure	闭包	entry	输入条, 条目
compile-time	编译时	exception	例外
composition	组合	executor	执行器
consistency	一致性	file descriptor	文件描述符
constructor	构造函数	file handle	文件句柄
CPAN (Comprehensive Perl Archive Network)	综合 Perl 文档网络	garbage collection	垃圾收集
CSV (comma-separated value)	逗号分隔值 (文件)	geometry management	布局管理
DBI (DataBase Interface)	数据库接口	glue code	粘连代码
declaration	(函数的) 声明	GUI (Graphic User Interface)	图形用户界面
delegation	代理	handle	句柄
denial of service	拒绝服务 (攻击)	hash	散列
dereference	间接引用	hash reference	散列引用
derived class	派生类	hash value	散列值
destructor	析构函数	hash variable	散列变量
directory handle	目录句柄	IDL (Interface Definition Language)	接口定义语言
dummy structure	哑结构	index	索引

indicator

指示器

instance

实例

introspection

内省

invocation

(函数的) 调用

interpreter

解释器

IPC (Interprocess Communication)

进程间通信

isolation

隔离性

item

元素

iterator

迭代器

life-cycle

生命周期

linker

链接程序

mark

标记

memory leak

内存泄漏

metadata

元数据

method

方法

MFC (Microsoft Foundation Class)

微软基础类库

module

模块

mutual recursion

相互递归

namespace

名字空间

non-blocking I/O

非阻塞 I/O

object

对象

ODBC (Open Database Connectivity)

开放数据库连接(标准)

OOP (Object-Oriented Programming)

面向对象编程

opcode

操作码

package

包

parser

分析器

persistence

持续性(存储)

pipe handle

管道句柄

POD (Plain Old Documentation)

普通旧式文档

pointer

指针

polymorphism

多态性

procedure

过程

process

进程

pseudo class

伪类

RDBMS (Relational Database Management System)	superclass
关系型数据库管理系统	超类
reference	symbol table
引用	符号表
reflection	syntax tree
反射	语法树
regular expression	table
正则表达式	表
RPC (Remote Procedure Call)	tag
远程过程调用	标签
RTTI (Run-time Type Identification)	taint check
运行时类型识别	污染检测
run-time	template
运行时	模板
safe compartment	thread
安全隔间	线程
scalar	tie
标量变量	绑定
scope	translator
作用域	翻译器
shadow class	trigger
影子类	触发器
sibling	typemap
兄弟 (节点, 指针)	类型映射
static	UIL (User Interface Language)
静态	用户接口语言
stream	virtual table
流	虚拟表
string	widget
字符串	组件
struct	wrapper
结构	包裹 (函数, 程序)
subroutine	zombie
子例程	僵尸 (进程)